UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

Aplicació de tècniques de models en temps d'execució i computació autònoma per a desenvolupar sistemes d'intel·ligència ambiental.

Un cas pràctic amb Kevoree.

Juliol de 2015

Màster en Enginyeria del Software, Mètodes Formals i Sistemes d'Informació.
Universitat Politècnica de València.
Departament de Sistemes Informàtics i Computació

Curs acadèmic 2014-2015.

Realitzat per:	Dirigit per:
Alberto Benetó Micó	Joan Fons Cors

Als meus pares i la meua germana.

"És el canvi, el canvi continu, el canvi inevitable,
el factor dominant de la societat actual."
Isaac Asimov.

Agraïments

A la vida ens creuem en moltes persones. Algunes d'elles arriben, romanen un temps, i se'n van. D'altres, estan per sempre amb nosaltres, siga al nostre costat o al nostre record. Siga com siga, sense aquestes persones no seríem el que som i, en el meu cas, aquestes persones teniu gran part de culpa de què jo puga haver portat aquest treball endavant.

Gràcies a Joan, el tutor d'aquest treball, per descobrir-nos al màster els temes sobre els quals tracta aquest treball i per saber transmetre l'entusiasme pel que fas i per l'ajuda i els ànims transmesos per realitzar aquest projecte.

Vull seguir donant les gràcies als que són els artífexs de què haja arribat fins ací. Del fet que puga dir molt orgullós, que tinc uns estudis i uns coneixements que ells no van tindre l'oportunitat d'assolir. Gràcies Pare i Mare per l'educació que ens heu donat, tant a mi com a Eva, perquè això és el que hui ens fa ser com som. Gràcies pels anys que heu sacrificat de la vostra vida, treballant de sol a sol i deixant de gaudir d'eixides i vacances per poder donar-nos una educació de qualitat. Açò ja acaba, ara us toca gaudir a vosaltres.

A la meua germana Eva, gràcies per ser la germana que eres. Saps que gran part de culpa de què jo em decantara per estudiar ciències és teua, i hui en dia estic molt content d'haver-te fet cas. Gràcies per ser una germana genial i, també, una amiga. I a més, segur que també seràs la millor mare que la meua neboda Daniela, que ja ve de camí, podrà tenir. Al meu cunyat Hèctor, que arribares fa pocs anys però ja eres un mes a la família. Segur que tu també seràs un pare genial, igual de genial que has sigut com a cunyat i amic. I a ma tia, que està "la mar de contenta" del seu nebot informàtic. Gràcies!

A Bea, per tots els anys que hem compartit. Per aguantar-me al llarg de tots aquests anys i ser la meua crossa quan les coses anaven malament, però sobretot per gaudir en mi les cosses bones que ens ha donat la vida. Perquè sense tu arribar fins ací no haguera sigut tan fàcil. Per tot el que has fet per mi, mil gràcies es queden curtes.

A Marc, que se'ns ha anat d'aventures a treballar al Japó. Per tots els moments bons que hem passat junts, pels no tan bons que hem sabut passar junts recolzant-nos l'un a l'altre i per tots aquests anys de vivències que hem compartit a Benimaclet. *Arigato* cosí!

Al Maria i Dani, que sempre heu estat ací, i sobretot aquests mesos que les coses han sigut més complicades. Que, juntament amb Víctor, sabíeu traure'm de casa quan em feia falta aire fresc (i quan no també). Per les històries que hem viscut junts, pels nostres moments al barri, i sobretot, per que sé que amb puc comptar amb vosaltres.

A la gent de la Universitat, Anna, Alba, Vivó, Moi, Riera, Carlos, Jordi, David, Jose, Jorge, Angel, Sandra, Dom, per tots els nostres anys entre classes, les hores d'estudi que hem passat junts, el nostre Athlètic, i els bons moments que he viscut amb tots vosaltres.

A la gent de GMV, Rubén, Bea, Adrián, Lucre, Miki, Enric, Javi, Carlos, Miguel, Barto i Inma. Sou un grup genial. Gràcies a vosaltres i a la vostra confiança he pogut créixer en un món laboral que no és tan fàcil com te'l mostren abans d'eixir de la Universitat. Sou uns grans professionals, i unes grans persones. Treballar amb vosaltres és un plaer dia rere dia. Gràcies per la vostra dedicació, per ensenyar-me tot el que sé i per la vostra comprensió durant el temps que he estat dedicant-me a realitzar aquest màster.

I a tots els que formeu part de la meua vida i no formeu part de les línies de dalt. No us puc citar a tots perquè sinó, no acabaria mai, però també, moltes gràcies.

Resum

En aquest treball de fi màster exposarem i treballarem sobre els principis de computació autònoma i auto adaptabilitat de sistemes aplicats a un escenari englobat en l'àmbit de la intel·ligència ambiental.

Volem definir un sistema capaç de respondre a diferents estímuls del entorn, capaç de autogestionar-se i auto adaptar-se a diferents casos i escenaris del sistema. Seguirem un enfocament orientat a models, i mes concretament a un enfocament orientat a models en temps d'execució. Amb açò, pretenem aconseguir un model que defineix fidelment el sistema i que es pot alterar sense interrompre el funcionament adequat d'aquest.

Sobre aquestes bases, pretenem generar un prototip amb un comportament simulat que ens permeta tenir un sistema ubic i que s'adapte al canvi, definit amb models que tenen el avantatge de poder ser modificats fàcilment en temps d'execució. Aquests defineixen completament el sistema, ens donen una visió d'alt nivell del que es, i ens faciliten la tasca de modelar i implementar tot el sistema. El avantatge que ens aporta és que podem modificar en temps d'execució aquest model, i aplicarho de forma que es plasme fidelment en el sistema.

Per a desenvolupar el nostre prototip presentarem Kevoree, una eina que ens permetrà desenvolupar , mitjançant tècniques model@run.time, el prototip d'un sistema aplicat a l'àmbit de la intel·ligència ambiental.

A més a més, detallarem la metodologia de treball hem seguit per realitzar els nostres desenvolupaments. Al món de la enginyeria del software, es vital l'organització i el seguiment de unes pautes i una metodologia clares, per poder orientar el nostre treball cap a la meta que volem aconseguir i poder assolir-la amb èxit.

Alberto Benetó
abeneto23@gmail.com

Joan Fons Cors
jjfons@dsic.upv.es

Abstract

In this final master paper, we will discuss and work on the principles of autonomic computing and self adaptability, by designing a little prototype of system into ambit of ambient intelligence.

We define a system able to respond to different environmental stimuli, capable of self-management and self-adapt to different cases and scenarios of the system. We follow a model-driven approach, and more particularly, a model-driven approach runtime. With this approach, we aim to get a model that accurately defines the system and can be altered without disrupting the proper functioning of the system.

On this basis, we intend to generate a prototype with a simulated behavior that allows us to have a ubiquitous system that adapts to change, defined with models that have the advantage of being easily modified at runtime. These fully define the system, give us a high-level view of what it is, and make easy to model and implement the system. The advantage that gives us is that we can change at runtime this model and apply it in a way that faithfully translated into the system.

To develop our prototype we use Kevoree, a tool that will allow us to develop, through model@run.time techniques, a prototype applied to the ambit of ubiquitous systems.

In addition, we will detail the methodology we followed for our developments. In the world of software engineering it is vital the organization and monitoring of guidelines and a clear methodology to guide our work toward the goal we want to achieve successfully.

Alberto Benetó
abeneto23@gmail.com

Joan Fons Cors
jjfons@dsic.upv.es

Resumen

En este trabajo de fin máster expondremos y trabajaremos sobre los principios de computación autónoma y auto adaptabilidad de sistemas aplicados a un escenario englobado en el ámbito de la inteligencia ambiental.

Queremos definir un sistema capaz de responder a diferentes estímulos del entorno, capaz de autogestionarse y auto adaptarse a diferentes casos y escenarios del sistema. Seguiremos un enfoque orientado a modelos, y más concretamente a un enfoque orientado a modelos en tiempo de ejecución. Con esto, pretendemos conseguir un modelo que define fielmente el sistema y que se puede alterar sin interrumpir el funcionamiento adecuado del mismo.

Sobre estas bases, pretendemos generar un prototipo con un comportamiento simulado que nos permita tener un sistema ubicuo y que se adapte al cambio, definido con modelos que tienen la ventaja de poder ser modificados fácilmente en tiempo de ejecución. Estos definen completamente el sistema, nos dan una visión de alto nivel de lo que es, y nos facilitan la tarea de modelar e implementar todo el sistema. La ventaja que nos aporta es que podemos modificar en tiempo de ejecución este modelo, y aplicarlo de forma que se plasme fielmente en el sistema.

Para desarrollar nuestro prototipo presentaremos Kevoree, una herramienta que nos permitirá desarrollar, mediante técnicas model@run.time, el prototipo de un sistema aplicado al ámbito de la inteligencia ambiental.

Además, detallaremos la metodología de trabajo hemos seguido para realizar nuestros desarrollos. En el mundo de la ingeniería del software, es vital la organización y el seguimiento de unas pautas y una metodología claras, para poder orientar nuestro trabajo hacia la meta que queremos conseguir y poder alcanzar con éxito.

Alberto Benetó
abeneto23@gmail.com

Joan Fons Cors
jjfons@dsic.upv.es

Índex

Índex de figures

Índex de codi font

Capítol 1

Introducció

En aquest treball de fi màster abordarem una gran quantitat de conceptes desconeguts per molts, però que són el futur i comencen a ser el present de la computació tal com la coneguem.

La computació ha avançat molt des de les primeres màquines de Turing, fins al punt de que hui en dia, a efectes pràctics en la vida de les persones, poc tenen a veure amb aquelles. De fet, un dels punts importants d'aquesta evolució se centra en la intercomunicació dels diferents dispositius que tenim al nostre abast per formar un tot que interactua i respon a les nostres peticions i necessitats. Hui en dia ja no ens interessen dispositius independents que facen tasques concretes, sinó que ens motiven els dispositius que són capaços d'interactuar amb altres per formar un sistema molt més complet i amb molta més funcionalitat i capacitat de resposta. Seguim lluny de sistemes súper intel·ligents com els que podem veure a les pel·lícules, però la tendència és clara: unir xicotets components o sistemes per formar un tot global que ens facilite les nostres tasques diàries. I tot açò, és innegable, ho ha fet possible Internet i la intercomunicació de xarxes.

Però açò ni és una cosa només a l'abast dels que estem clavats en aquest món ni està tan lluny com pareix. De fet hui en dia ja la majoria de sistemes no estan formats per un sol dispositiu o una sola maquina, sinó que solen estar formats per diferents dispositius i sistemes que interactuen per formar un tot. I normalment, aquests sistemes està dispersos, i ja comencen, per dir-ho d'alguna forma a "camuflar-se" de forma que no esperen la interacció directa de l'usuari, sinó que s'anticipen a aquest, de forma que l'usuari ix del bucle pregunta/resposta i és el sistema el que actua i respon. Açò és bàsicament el que s'espera d'un sistema ubic.

A més a més, volem que aquests sistemes puguen auto adaptar-se o canviar-se sense necessitat d'intervenció del factor humà. És el que coneguem com a sistema auto adaptatiu.

El que bàsicament veurem en aquest treball de fi de màster és això, un sistema capaç de funcionar per ell mateix, sense necessitat d'intervenció del factor humà, capaç d'integrar diversos sistemes per formar un sol, capaç d'interactuar entre els diferents subsistemes i components que el formen i capaç d'auto adaptar-se a canvis, problemes o peticions externes que puguen sorgir.

A més a més, entra un joc un altre factor molt important i que serà l'eix principal sobre el qual girarà el nostre treball: la utilització de models per definir i implantar aquests sistemes. Açò ens permetrà obtindre un altíssim nivell d'abstracció i la possibilitat de modificar aquests sistemes en temps d'execució únicament alterant el o els models que el representen.

Per a la implantació del nostre prototip utilitzarem una eina que actualment segueix en desenvolupament, però que ja dona a entreveure que serà una eina molt potent, anomenada Kevoree. Kevoree es basa en el principi del Model@Run.time, és a dir la utilització de models per definir sistemes i la possibilitat d'alterar-los en temps d'execució. Kevoree també ens facilita la tasca de treballar amb sistemes distribuïts, de forma que podem tindre diferents sistemes representats per diferents models en nodes separant que interactuen entre si de forma senzilla, utilitzant únicament els models per definir aquests sistemes i la intercomunicació entre ells.

La unió d'aquest enfocament Model@Run.time amb els conceptes de computació ubiqua i autoadaptabilitat dels sistemes ens permetrà obtindré un sistema autosuficient, i fàcilment alterable.

1.1 Problematica i motivació

Els sistemes auto-adaptatius, és a dir, sistemes que són capaços de canviar i adaptar-se a les condicions variants de l'entorn són cada vegada més demandats. Cada vegada tendim més a aquest tipus de sistemes, de forma que la intervenció humana siga cada vegada menys necessària. Per exemple hui en dia, ja tenim servidors virtuals que són capaços d'autoadaptar-se a les condicions variants de l'entorn augmentant-se la memòria o reduint-se-la, incrementant o disminuint el nombre de processadors segons la càrrega d'un portal web, per exemple. I així, diversos sistemes.

Aquests sistemes però, són sistemes normalment molt grans i complicats de veure o tractar a primer cop de vista. Necessitem adoptar tècniques per facilitar totes aquestes tasques. És evident que, des de la coneguda com Crisi del Software, les

formes de desenvolupar programari han evolucionat molt, i encara que la nostra encara és una ciència en construcció, pel que fa a metodologies i formes de treball, ja comptem amb tècniques i metodologies que ens marquen una línia de bones pràctiques a seguir. Açò és molt important en els casos com el que hem comentat, en el que el sistema és molt gran o complex. I és ací on creguem que la utilització de models juga un paper fonamental.

Els models han canviat la visió dels desenvolupaments dels sistemes software. Quan un sistema comença amb un model, tot canvia. Tenim, com a punt de partida, una visió simple i directa d'un sistema que potser molt més complicat. Podem fer un canvi de forma bastant més intuïtiva i simple, i a més a més, aquests models incorporen tècniques de transformació a codi que ens poden estalviar molta feina a l'hora de passar a la implementació física del sistema. En el següent apartat comentarem detalladament en què consisteix el desenvolupament orientat a models i quins avantatges ens ofereix aquesta forma de desenvolupament d'un sistema software. Però per fer un xicotet avançament, podem afirmar que els models ens aporten grans avantatges a l'hora de definir i implantar un sistema software:

- Millora la productivitat del desenvolupament
- Dona la possibilitat d'automatitzar el procés de desenvolupament (existeixen tècniques de transformació de model a codi).
- Redueix el nombre de defectes en el codi (els models permeten una validació primerenca del sistema).
- Facilita la comprensió i documenta el sistema. (possibilitat també de generar aquesta documentació de forma automàtica).
- Millora la descomposició i modularització del software.
- Facilita l'evolució i manteniment del programari.
- Millora la reutilització.
- Facilita, en cas de sistemes ben modelats, l'escalabilitat del software.

La problemàtica que se'ns plateja gira al voltant dels sistemes autoadaptatius i la seua integració amb el desenvolupament orientat a models. Es pretén implantar un sistema que siga capaç de respondre als estímuls de l'entorn i adaptar-se al seu àmbit i als canvis que es produïsquen en aquest sentint utilitzant un enfocament orientat a models. Vist així, pareixen conceptes que no tenen res a veure a primer cop de vista, però que si ho pensem bé té molt de sentit. És molt més pràctic, còmode, i sobretot, dona menys peu a què es produïsca una condició d'error, que aquests sistemes autogestionables puguen produir canvis en ells mateixos mitjan-

çant un model que directament modificant el que són els components que formen el sistema.

És a dir, ja no pretenem fer canvis directament al sistema. El que pretenem és modificar el model que defineix aquest sistema en temps d'execució i que el nostre sistema siga capaç d'adaptar-se automàticament a aquest canvi de model sense haver de detindre's. És a dir, en temps d'execució. És més, en veritat anem un poc més enllà i pretenem que siga el mateix sistema el que provoque aquests canvis en el model, convertint-se en un sistema totalment independent i deslligat de la nostra col·laboració per produir un canvi o arreglar una situació d'error.

1.2 Objectius del treball

Dins de l'àmbit dels sistemes autoadaptatius pretenem buscar una solució que es recolze amb el treball amb models per definir i implantar un sistema simple i que, a més a més siga capaç, no sols de què nosaltres podem modificar-lo en temps d'execució, sinó que siga el mateix sistema el que siga capaç de reaccionar a estímuls i autoadaptar-se i generar aquests canvis en el model que posteriorment, seran aplicats al sistema per obtindre un sistema diferent al qual teníem en primera instància.

Per açò, anem a treballar en un prototip basat en un subsistema d'una ciutat intel·ligent.

Treballarem en aquest àmbit perquè el tema de les ciutats intel·ligent és un aspecte novedós del que és el dia a dia a les nostres vides.

Les ciutats intel·ligents són anomenades així per moltes coses, però una d'elles, és perquè són o deuen ser capaces d'autoadaptar-se i automodificar-se depenent del que passe dins d'ella en cadascun dels subsistemes que formen el sistema global de la ciutat. És per això que creguem que aquest és un escenari suficientment il·lustratiu per realitzar un xicotet prototip.

És a dir, com a objectiu tenim la realització d'un prototip d'un sistema en l'àmbit d'una ciutat intel·ligent, que siga capaç de ser definit amb models i que aquests models es puguen modificar en temps d'execució tant de forma manual com automàtica, donant suport al tema de la autoadaptatibilitat dels sistemes o computació autònoma.

El que pretenem demostrar que Kevoree és una ferramenta vàlida per a realitzar el modelatge de qualsevol tipus de sistema. Ens hem decantat per un escenari basat en una ciutat intel·ligent per decisió personal però seria també útil en altres escenaris. Per exemple, per modelar un entramat de servidors, els quals deuen respondre i autoescalar-se depenent de les peticions dels usuaris externs.

Com a punts principals del nostre treball, pretenem assolir els següents objectius:

- Definir un prototip d'un sistema simple en l'àmbit de la intel·ligència ambiental
- Donar-li al sistema un comportament simulat de forma que podem emular esdeveniments externs
- Fer que el nostre prototip siga capaç de reaccionar a aquests estímuls externs
- Definir aquest sistema amb models, mes en concret, modelant-lo amb el framework Kevoree.
- Mostrar com, amb Kevoree, podem produir canvis aquest model en temps d'execució.

1.3 Solució proposada

Per desenvolupar aquest prototip, necessitem utilitzar alguna eina que ens ajude a fer aquest treball. No té cap sentit començar a fer-ho tot des de zero, ja que estaríem saltant-nos un dels punts bàsics i principal del treball amb models: la reutilització. A part d'açò, també volíem trobar algun tipus de framework o programari que ja poguérem utilitzar per veure quin és l'abast real d'aquest tipus d'eines.

És per això que necessitem trobar una eina en la qual els models siguen la base per al desenvolupament i posada en marxa dels seus sistemes. Una eina que a més de permetre'ns modelar eixos sistemes, ens òbriga la porta a poder modelar sistemes que puguen produir canvis en la seua estructura modificant els models que el representen sense la nostra ajuda. És a dir, l'eina ens ha de permetre implantar via models sistemes autoadaptables sota la filosofia del modelatge en temps d'execució.

Com ja hem anomenat en el punt anterior, l'eina que hem escollit per fer açò s'anomena Kevoree. Kevoree planteja com a base el treball en models i sobretot el treball en models en temps d'execució. És un sistema encara en desenvolupament i open source, per això hem decidit explotar-lo per veure que és en veritat el que ens permet fer aquest sistema, fins on ens deixa arribar.

Amb l'ajuda de Kevoree dissenyarem un prototip d'un sistema basat en el paradigma de la intel·ligència ambiental i en concret, dins de l'àmbit de les ciutats intel·ligents, ja que creguem que és un entorn suficientment canviat per servir d'il·lustració del que pretenem aconseguir. Per fer açò Kevoree ja ens oferix una sèrie d'executables per posar en marxa l'entorn de forma simple, i un editor per poder començar a treballar amb el model generat pel runtime de forma immediata.

En capítols posteriors mostrarem fins on pot arribar Kevoree però podem avançar que, a pesar de tindre mancances encara, pel fet que seguix en desenvolupament, és una eina molt potent pel treball amb models i que dóna una gran facilitat a l'hora de realitzar treballs que impliquen modificacions de models en temps d'execució.

Destacar també que en el treball que ens ocupa, intentarem utilitzar Kevoree per a realitzar un treball en els àmbits que ja hem comentat (autoadaptabilitat, autogestió, ubiqüitat...) però si el que volem és simplement implantar un sistema més aviat senzill (per exemple, un xicotet servidor RESTful) d'acord amb un model, Kevoree ens ho va a permetre també sense problemes. És més, com veurem més avant, segurament podem arribar a alçar-lo sense haver de desenvolupar absolutament res, ja que té una llibreria de components molt completa. Però no de canals, tema sobre el qual aprofundirem un poc mes.

Seguidament, detallarem quina serà l'estructura del treball que estem presentant.

1.4 Estructura del treball

L'estructura que anem a seguir en aquest treball serà la següent: Primerament, oferirem una xicoteta visió de l'estat de la qüestió en aspectes bàsics del nostre sistema, com són la computació autònoma dirigida a sistemes auto-adaptatius.

Seguidament, parlarem de l'estat de la qüestió en temes de intel·ligència ambiental, ja que el nostre prototip va a centrar-se en aquest àmbit.

També parlarem de l'estat de la qüestió del modelatge o treball amb models per definir sistemes: que és, perquè servix, quins avantatges i desavantatges ens suposa el treball amb models, tècniques de modelatge i el tractament i modificació de models en temps d'execució.

Seguidament aprofundirem en l'eina que hem emprat per desenvolupar el nostre prototip: Kevoree. Parlarem de com funciona l'eina, com està estructurada, com modelar un sistema bàsic, com desenvolupar components nous o fins i tot com funcionen les tècniques de modelatge en temps d'execució en Kevoree i com dissenyar un component per què siga autoadaptable gràcies al treball en models Kevoree.

Per últim, exposarem el nostre prototip. Assentarem les bases de l'exemple que volem desenvolupar, comentarem com volíem fer-ho, com ho hem pogut fer, i quines coses bones i quines limitacions ens hem trobat a l'hora d'emprar Kevoree com a eina per al nostre treball.

Per finalitzar, plantejarem una sèrie de conclusions que hem extret del treball i deixarem la porta oberta a possibles treballs futurs.

A l'apèndix comentarem la metodologia que hem utilitzat per realitzar els desenvolupaments que hem necessitat per donar-li forma al nostre prototip. Al món de l'enginyeria del software, seguir adequadament una metodologia o no pot suposar l'èxit o el fracàs en un desenvolupament. En el cas que ens ocupa, el nostre desenvolupament no és una cosa critica ni que genere molts artefactes, ni que necessite una recollida prèvia excessiva de requisits. Així i tot, hem intentat seguir algunes pautes de la metodologia SCRUM, sobretot en quan a gestió de versions, evitant algunes fases d'aquesta, sobretot les mes rel·lacionades en la gestió i seguiment del procés, per fer més lleuger el compliment d'aquesta, però donant-li molta importància al versionat i backup del codi.

Capítol 2

Computació autònoma, intel·ligència ambiental i treball amb models

2.1 Introducció

El treball que anem a presentar gira sobre uns elements fonamentals que seran els pilars de la nostra recerca. El que busquem en aquest treball és aprofundir en diferents àmbits que ara anem a anomenar.

Per una banda, considerem que la **computació autònoma** és un tema molt interessant, sobre el qual s'ha de seguir avançant. Cada vegada és més necessari que els nostres sistemes puguen tindre cert nivell d'independència. Aquests sistemes són capaços **d'autogestionar-se** , per superar la complexitat que suposa una **adaptació ràpida i precisa** a les condicions canviants de l'entorn o del mateix sistema, i reduir la barrera que aquesta complexitat planteja al creixement addicional.

Per altra banda, ens interessa molt aprofundir sobre el treball amb sistemes orientats a models (**M**odel **D**riven **E**ngineering). Aquest paradigma comença a prendre força durant els últims anys i comença a ser una alternativa forta i robusta per a tindre en compte a l'hora de començar un procés de desenvolupament software. Aquest enfocament té com a objectiu, entre d'altres, simplificar el procés de disseny i implementació d'un sistema, gestionar de forma adequada la creixent complexitat dels sistemes i fomentar la re utilització.

El treball amb models ens pareixia molt interessant, però encara més el seu vessant, la utilització dels models en temps d'execució. Aquests no sols ens ofereixen els avantatges que ens dóna el desenvolupament orientat a models, sinó que a més, ens dóna l'avantatge de poder treballar amb aquests models mentre el sistema que aquesta representant està en marxa.

Per últim, parlarem també de l'estat de l'art de la intel·ligència ambiental, ja que el nostre prototip se centrara en aquest escenari.

El que pretenem és unir tots aquests punts, de forma que desenvoluparem un prototip d'un sistema simulat en l'àmbit de la intel·ligència ambiental, que tinga la capacitat del treball autònom i l'autoadaptabilitat, a més de dissenyar e implementar tot açò amb l'ajuda de models i de frameworks que ens ajudaran a treballar amb aquests models en temps d'execució. Gràcies a l'auto adaptabilitat del nostre sistema i a la seua autonomia, serem capaços de produir els canvis necessaris directament al model, aconseguint així la autoadaptabilitat i l'autonomia del nostre sistema via la modificació del model del nostre sistema en temps real.

2.2 Computació autònoma

Hui en dia, milions de sistemes, milions de persones, milions d'aparells, romanen connectats i funcionant donant forma a un "mon paral·lel"d'interconnexió de sistemes i muntons de dades que naveguen de punta a punta del planeta. Aquests sistemes cada vegada son mes i més complexos a causa de les exigències i necessitats de negocis, persones, estats...

La quantitat i la complexitat d'aquests sistemes provoca una gran necessitat d'experts que mantinguen i corregisquen aquests sistemes en cas de fallades o necessitats de canvis a causa de les condicions canviants de la societat en què vivim. A més, aquest "problema"va en augment, és a dir, els sistemes van a tendir a ser cada vegada més complexos i a necessitar cada vegada més canvis, millores i correccions a causa del frenètic ritme de vida de la societat tecnològica en la qual vivim.

És necessari, doncs, que aquests sistemes es tornen cada vegada més intel·ligents, intentar que cada vegada depenguen menys de l'ésser humà i comencen a tindre destreses pròpies, com aprendre i poder reaccionar davant d'estímuls externs i interns del sistema.

En 2001, Paul Horn, Vicepresident d'IBM, va proposar una solució: Sistemes de computació, que s'auto regulen, de la mateixa forma que el sistema d'un ésser humà regula i protegeix la nostra integritat. A aquesta primera descripció de cap on hauria d'evolucionar un sistema se la va encunyar com **computació autònoma**.

La computació autònoma ha evolucionat cap a una disciplina per crear sistemes i aplicacions software que siguen capaces d'auto gestionar-se i auto adaptar-se per intentar superar les dificultats creixents per mantenir un sistema amb una alta complexitat computacional. (Cetina Englada 1998)

IBM, a més, en 2001 també, va publicar un manifest en el qual els sistemes es comparaven amb el cos humà, és a dir, un sistema complex però amb un sistema (en aquest cas el nerviós) que comptava amb autonomia per controlar totes les funcions del cos, i proposà treballar cap a sistemes computacionals que comptaren, salvant les distàncies, amb la mateixa capacitat.

2.2.1 Propietats dels sistemes autònoms

IBM plantejà quatre propietats que un sistema autònom o auto-adaptatiu havia de tindre: **auto configuració**, **auto optimització**, **auto reparació** i **auto protecció** (Gómez Lacruz 2011):

- **Auto configuració**. D'un sistema autònom s'espera que siga capaç d'auto configurar-se depenen objectius d'alt nivell, és a dir, especificant el que es desitja aconseguir, no el com es va a aconseguir.
- **Auto optimització**. D'un sistema autònom s'espera que siga capaç d'optimitzar els seus recursos de forma eficient. Ha de ser capaç de provocar un canvi en el sistema si és necessari amb l'objectiu de millorar el rendiment o la qualitat del servei que ofereix.
- **Auto reparació**. D'un sistema autònom s'espera que siga capaç de descobrir, diagnosticar i reparar els components defectuosos sense haver d'interrompre els serveis que ofereix. Ha de garantir, també, que els sistemes o serveis crítics amb què compte no van a trencar-se o parar-se. Per poder aconseguir açò, aquests sistemes han de tindre la capacitat d'auto recuperació d'una situació inesperada.
- **Auto protecció**. D'un sistema autònom s'espera que siga capaç d'anticipar, detectar, identificar i protegir contra qualsevol tipus d'atac. Seran capaços de protegir-se tant d'atacs externs com d'atacs provocats per fallades en cascada a causa d'errades en mesures d'auto protecció. També ha de ser capaç d'anticipar-se als problemes d'acord amb informes de sensors integrats al sistema i prendre mesures per vigilar-los o corregir-los abans que provoquen un estat d'error del sistema.

2.3 Models com a base dels sistemes

Altre dels punts principals del nostre treball és l'enfocament que se li dóna a un sistema si és definit via models.

Actualment, el Desenvolupament Dirigit per Models (MDD) o l'Enginyeria Dirigida per Models (MDE) és un paradigma cada vegada més estés de desenvolupament de software que defén que les aplicacions s'han d'especificar amb un alt nivell d'abstracció usant models. Açò no implica que, posteriorment, hagem de llegir i adaptar eixos models a codi o a l'aplicació física final sinó que, mitjançant tècniques de transformació, es poden generar implementacions d'aquest model, i molts altres artefactes com per exemple, la documentació associada al sistema.

Amb l'acollida del desenvolupament dirigit per models per al desenvolupar sistemes software, obtenim una sèrie d'avantatges. Una d'elles és la capacitat de separar el domini del sistema de la implementació tecnològica d'aquest. A més, els models formals i les seves transformacions permeten obtenir un alt grau de traçabilitat entre models i codi, de forma que obtenim una correspondència directa entre el model i els requeriments del sistema, el disseny, la implementació i les proves (Gómez Lacruz 2011). És a dir, simplement amb un bon modelat del nostre sistema podem incrementar exponencialment la productivitat i la qualitat del nostre software.

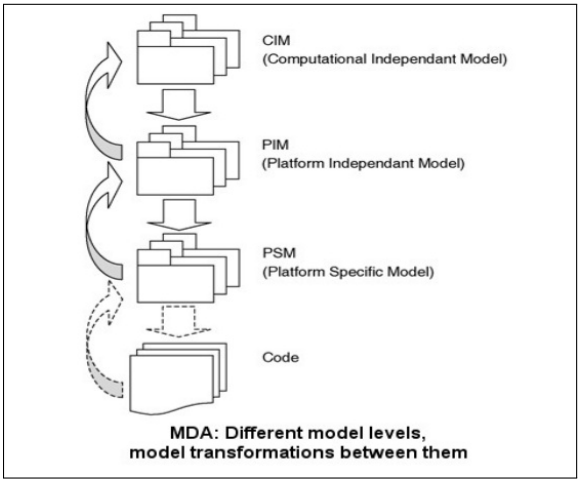

Figura 2.1: Procés de transformació de models (Cabot 2011)

Per poder portar a terme tot açò de forma pràctica, existeixen diverses ferramentes i frameworks per portar-ho a terme. Els passos bàsics en els quals va a contar un procés MDD va a consistir en un conjunt de transformacions M2M (model a model) més una transformació final M2T (model a text) (figura 2.1). Aquest procés compta amb diferents nivells de models, començant per **CIM** (Model computacional independent), a continuació trobem **PIM** (Model independent de la plataforma), seguidament tenim els models **PSM** (Models específics de la plataforma) i per últim ja obtenim la transformació al nostre codi font.

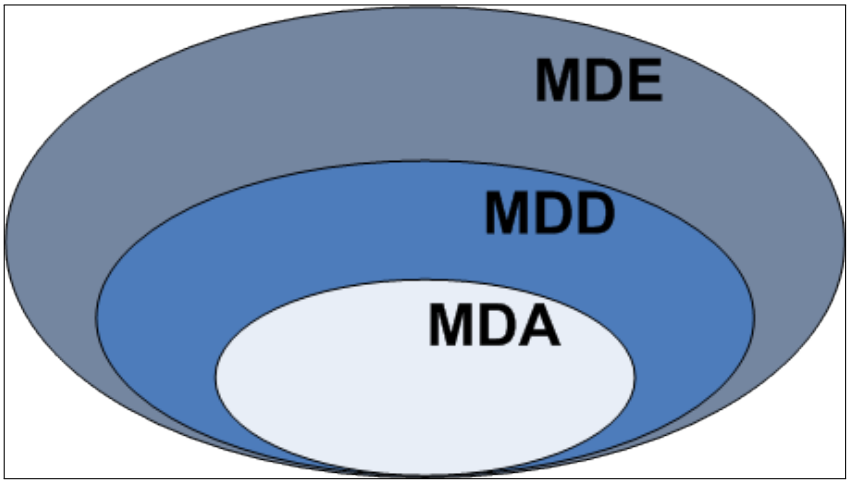

Figura 2.2: Els diferents nivells dels que parlem quan ens referim al modelatge de sistemes: Model-driving engineering, model-driven development i model-driven architecture

2.4 Cicle de vida del desenvolupament dirigit per models

Quan parlem del desenvolupament dirigit per models, en veritat no estem parlant de que sols tenim un model i ja. No, de fet, MDD basa la seua funcionalitat en tres tipus diferents de models:

- CIM (Model computacionalment independent) : Aquest model es una representació del sistema que no recull detalls de l'estructura d'aquest. També se'l coneix com model de domini. Aquest model sol associar-se a les fases d'anàlisi i de recollida de requisits en un desenvolupament software.
- PIM (Model independent de la plataforma): Aquest model s'especifica mitjançant un llenguatge especific de domini, però independent de qualsevol tecnologia.

- PSM (Model especific de la plataforma): Aquest model es pot obtenir de forma automàtica aplicant transformacions sobre els models anteriors. De fet el PMI pot ser traduït a mes d'un model PSM.

Una vegada tenim definits cadascun d'estos models, podem obtindre el codi font a partir de cada PSM que hem obtingut.

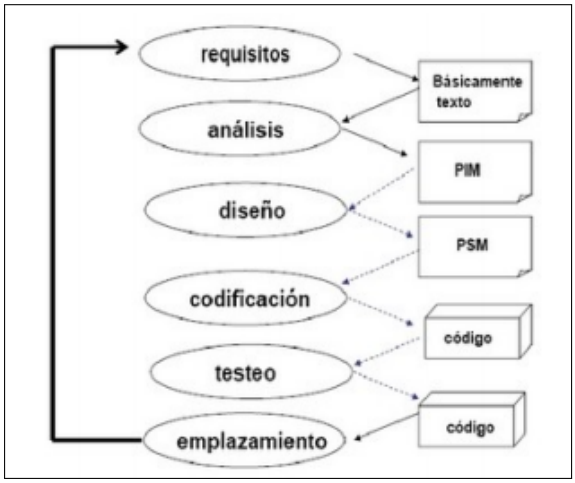

Figura 2.3: Cicle de vida del desenvolupament dirigit per models. (Pons, Giandini i Pérez 2010)

2.5 Models en temps d'execució

Els models, com ja hem vist, ens aporten grans millores a l'hora de desenvolupar software de qualitat, i sobretot, ens dóna gran facilitat a l'hora de permetre i aplicar canvis al nostre sistema. Canvis que, per altra banda, cada dia són més constants i necessaris quan tenim un sistema software en funcionament que ha d'atendre una creixent càrrega de treball o ha d'adaptar-se a la velocitat de canvi del món que ens envolta (Morin et al. 2009). Ara bé, a la majoria de sistemes no podem contemplar una parada del sistema per provocar un canvi en aquest. És més, de vegades és el mateix sistema el que es té que auto adaptar (com ja hem vist a l'apartat anterior). És ací on cobren protagonisme els models en temps d'execució.

L'objectiu de les tècniques model@run.time és estendre els avantatges del treball en models a les fases d'execució del sistema. L'aproximació model@run.time té algunes diferències amb MDE. Podem afirmar que, un model en temps d'execució

és una representació fidel dels sistemes associats a aquest que remarca l'estructura, els objectius o el comportament d'un sistema dins del context d'un àmbit o problema.

Un aspecte important per al nostre treball és que els models en temps d'execució són considerats com una tecnologia clau per als sistemes que es controlen a si mateixos, és a dir, per als sistemes auto adaptatius. Aquests models en temps d'execució es poden utilitzar per diferents objectius: per **controlar i motoritzar** sistemes durant l'execució, per suportar la **integració semàntica** d'elements heterogenis de software en temps d'execució, per **corregir errors de disseny**, per plasmar al sistema durant la seua execució **noves decisions de disseny**, i per donar suport a **canvis inesperats** i a la **evolució dinàmica** de disseny del software.

Hem de tindre un compte que modelar un sistema requereix temps i esforç, i que no es pot modelar tot. Hem d'aconseguir fer una aproximació prou fidel perquè plasme exactament les propietats, arquitectura i funcionament del nostre sistema però sense aprofundir en aspectes no rellevants d'aquest. Un bon modelat ens permetrà aplicar correctament les tècniques model@Run.time. L'esforç per modelar el sistema durant la fase de disseny és útil tant per produir el nostre sistema com per a ser una base fidel que posteriorment ens donarà l'opció de canviar en temps d'execució, a més que un sistema ben modelat té un gran avantatge pel que fa a la seua autoadaptació.

En el nostre treball, el modelatge en temps d'execució serà la base per què els nostres sistemes puguen auto adaptar-se, ja que per poder aconseguir-ho, el nostre sistema provocarà **canvis de model** i no canvis als components o al codi del programa directament.

Seguidament passarem a descriure el que coneguem com intel·ligència ambiental, escenari en el qual hem posicionat el nostre prototip.

2.6 Intel·ligència ambiental

Actualment, la forma de comunicar-nos amb els dispositius que ens envolten o que formen un sistema és més aviat directa. És a dir, esperem una resposta quan nosaltres provoquem eixa resposta.

Però hi ha situacions en les quals, eixa resposta, el sistema podria ser capaç d'oferir-nos-la sense rebre cap estimul nostre. Per exemple, si jo entre a l'ascensor cada dia a les set del matí, aquest ascensor podria ser capaç de saber que el meu destí és la planta zero, com cada dia, sense necessitat del fet que jo polsara el botó.

Tot va néixer a mitjan anys 80 a les instal·lacions del Xerox Parc Lab de Califòrnia. El precursor de què ara coneguem com a intel·ligència ambienta fou Mark Weiser. Weiser va suggerir la desaparició de la computadora actual tal com la coneguem i la distribució de la computació en xicotets dispositius de funcionalitats reduïdes i omnipresents en l'entorn que ens rodeja. Com a curiositat, a Weiser també si li atribueix el tractament de la distòpia en la novel·la Ubik de Philip. K. Dick, en la qual podíem entreveure un futur en el qual absolutament tot estaria interconnectat.

Si haguérem de definir la intel·ligència ambiental (o ubiqüitat) d'alguna forma, podríem dir que és la integració de la computació en l'entorn de l'ésser humà de forma que els elements computacionals que formen aquest entorn no s'aprecien com objectes diferenciats. És a dir, ho podem considerar com un paradigma en què les persones estan immerses en un espai digital que és conscient de la seua presencia, s'adapta a les seues necessitats, a les seues costums, i és sensible al context. (Martínez Acuña 2013)

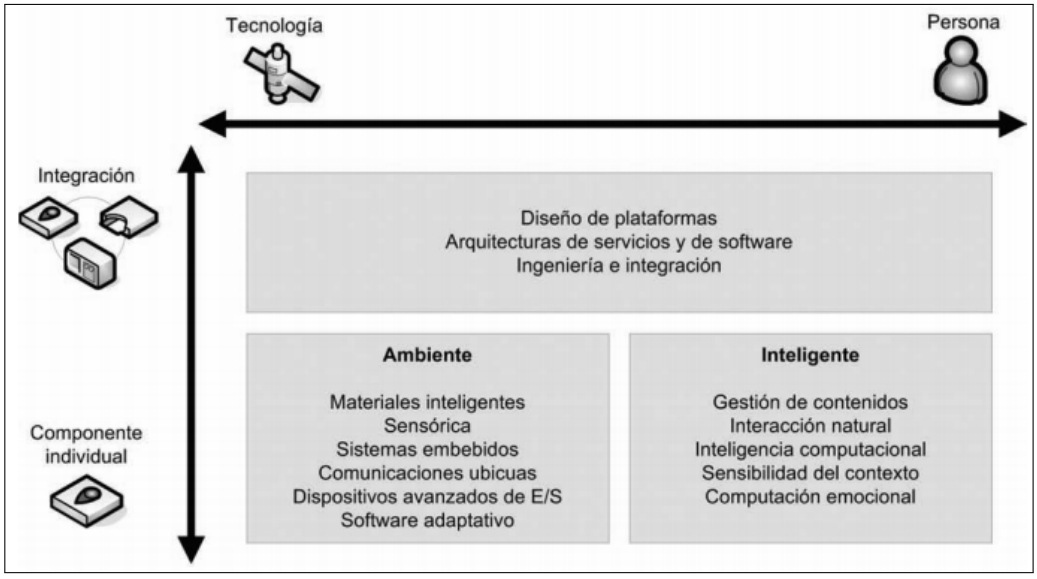

Figura 2.4: Eixos de desenvolupament de la Intel·ligència Ambiental (Martínez Acuña 2013)

Al final, si ho haguérem de definir per a una persona que no entén del tema que estem tractant, un sistema d'intel·ligència ambiental seria aquell en el qual no es necessita una interacció directa o proactiva de l'usuari de cada al sistema, sinó que és el sistema el que s'encarregarà de satisfer les nostres necessitats adaptant-se a nosaltres, a l'entorn, i canviant si és necessari.

Aquesta capacitat de canvi és la que volem facilitar amb el treball amb models en temps d'execució, que hem comentat en l'apartat anterior. I és per això, perquè és un escenari on la capacitat de canviar, d'autoadaptar-se és molt important, que hem vist aquest escenari l'ideal per definir un senzill prototip per mostrar la utilitat del treball amb models en temps d'execució.

En concret, és en l'àmbit de les ciutats intel·ligents on hem situat el nostre prototip. Una ciutat intel·ligent és un tipus de desenvolupament humà basat en la sostenibilitat, capaç de respondre a necessitats d'aquells que l'habiten (persones, empreses, institucions...) en qualsevol aspecte operatiu, social i ambiental.

Evidentment, per aconseguir tot açò, necessitem tindre una forta càrrega d'intel·ligència ambiental en la ciutat: necessitarem molts sistemes que interactuen entre si per formar un sistema global que serà la nostra ciutat. Aquests sistemes (o millor dit, subsistemes) deuran ser invisibles als habitants de la ciutat, de forma que s'encarregaran d'atendre les necessitats dels habitants sense que aquests hagen d'interactuar de forma pro activa amb cadascun dels subsistemes. A més, si algun d'aquests subsistemes necessita canviar o adaptar-se a un canvi en el seu entorn, deurà ser capaç de fer-ho.

Un exemple d'aquest tipus de ciutat és New Songdo City. Aquesta és una ciutat que està ubicada a uns 60 kilòmetres a l'oest de la ciutat sud-koreana de Seúl. Ací tots els sistemes que formen la ciutat estaran interconnectats i els computadors o elements computacionals estaran integrats als habitatges, carrers i edificis d'oficines. (Kuecker 2013)

No coneixem a fons els sistemes que integra aquesta ciutat però un d'ells podria ser el que plasmarem al nostre prototip: un sistema que, amb l'ajuda de drons, puga controlar el transit, el correcte funcionament de les vies de comunicació, que no es produïsquen infraccions. I que donat el cas, per exemple, per un accident, es puga **canviar el model de la ciutat**, és a dir, per exemple, canviar la funcionalitat normal dels semàfors per donar eixida al transit.

Aquest és l'objectiu que perseguim al nostre treball, és a dir combinar conceptes com la computació autònoma, l'autoadaptabilitat i el treball amb models en temps d'execució amb l'eina Kevoree que més avant veurem, dissenyar un xicotet prototip en aquest àmbit, el de les ciutats intel·ligents i la intel·ligència ambiental.

Capítol 3

Kevoree

El nostre treball gira al voltant d'una eina anomenada Kevoree. Joan em va recomanar utilitzar i investigar sobre aquesta eina, ja que ha evolucionat els últims anys i ens ofereix una solució ràpida per implantar sistemes basats en tècniques de modelatge en temps d'execució. A més a més Kevoree ens ofereix solucions tancades de components ja desenvolupats i ens facilita la re utilització de codi, evitant nous desenvolupaments innecessaris i que ens han aprofitat per plantejar el nostre cas pràctic. Presentarem que és Kevoree, mostrarem com aplica tècniques de modelatge en temps d'execució, presentarem la gran diversitat de sistemes que ens permet implantar, mostrarem que ens ofereix Kevoree i per últim mostrarem com podem desenvolupar els nostres propis components (elements del model) per utilitzar-los en el modelatge del nostre propi sistema.

La majoria dels coneixements assolits sobre Kevoree els hem pogut obtenir gràcies tant a la documentació (Fouquet,François et al. 2015b) de Kevoree que ha publicat l'equip que treballa en el seu desenvolupament com als xicotets tutorials (Fouquet,François et al. 2015a) que han publicat també aquest mateix equip. Tal volta hem trobat a faltar un poc mes de documentació però com a punt de partida és suficient per començar a treballar amb Kevoree.

3.1 El framework Kevoree

Kevoree és una eina, o més ben dit, un conjunt de ferramentes que ens van a permetre definir el model d'un sistema i fer que aquest model es transforme en un sistema real i funcional. Kevoree és un projecte OpenSource desenvolupat per François Fouquet i Erwan Daubert.

Figura 3.1: Logotip del projecte Kevoree

El que ens planteja Kevoree, o les premisses sobre les quals s'asseu Kevoree són les següents:

- Permetre el desenvolupament de programari reconfigurable, amb cert enfocament als sistemes distribuïts.
- Aprofitar l'enfocament Model@run.time per a oferir eines per construir, adaptar i sincronitzar els sistemes distribuïts. (en veritat, encara que el nostre projecte és distribuït, no és necessari que qualsevol implementació en Kevoree ho siga)
- Multiplataforma. Com que està desenvolupat en Java, es pot utilitzar en diferents plataformes, un ordinador personal, servidor, Raspberry...
- Kevoree ofereix una visió senzilla del modelatge, amb diferents components com nodes, canals de comunicació...
- Escalabilitat. Ja que el sistema es pot comportar de forma autoadaptable, canviant el model en temps d'execució, ens ofereix l'avantatge de poder escalar el sistema depenent de les necessitats d'aquest o dels usuaris.

Sobre el paper, i una vegada emprada l'eina per desenvolupar el tema del nostre projecte podem afirmar que compleix moltes d'aquestes característiques, encara que s'aprecia un nivell encara menut de maduresa, per altra banda propi d'una eina encara en desenvolupament.

El gran avantatge que creguem que presenta és la facilitat d'ús i la facilitat quant al modelatge i la re-utilització de codi. Kevoree ofereix components ja desenvolupats que ens han ajudat en certa mesura a realitzar els nostres desenvolupaments i a configurar un model estable. Com a punts dèbils podem destacar la part de comunicació entre nodes. Al nostre projecte necessitàvem comunicar dos models diferents, en diferents màquines, i ha sigut una tasca pràcticament impossible.

Altre dels punts dèbils, que també denota com anteriorment hem comentat, un baix nivell de maduresa, és la falta de documentació de la majoria d'elements del framework, cosa que ha dificultat en gran mesura el desenvolupament que hem realitzat dels nostres components.

A continuació descriurem el funcionament bàsic de Kevoree, els elements que el formen, com podem definir un model amb Kevoree i quines implicacions té açò i per últim la part que ens interessa, com podem modificar aquests models en temps d'execució per a què aplique els canvis directament al nostre sistema.

Com a nota important volem destacar la dificultat extra que ens ha suposat comprendre com funciona Kevoree, com està muntat i per a què serveixen cadascun dels components que el formen. Açò s'ha degut a la falta de documentació bàsica, o mínima, ja que de vegades era inexistent. Però hem d'agrair, això si, als desenvolupadors, que sempre que els hem plantejat qualsevol dubte mitjançant GitHub ens han contestat de forma bastant ràpida i de la millor forma possible. Si hem aconseguit arribar a algun lloc en gran mesura és gràcies a ells.

3.2 Prerequisits

Com qualsevol entorn, es necessiten uns elements bàsics per poder posar en funcionament tot el sistema. Kevoree està desenvolupat sobre Java i treballa sobre Java. De fet Kevoree és una adaptació d'una màquina virtual Java (amb les seues peculiaritats).

Per poder treballar amb Kevoree necessitem tindre instal·lada mínimament una JRE Java. Quan començarem a treballar amb Kevoree, la versió Java que gastàvem era la 7, però un dia de sobte Kevoree deixà de funcionar. Des d'aquest moment, Kevoree comença a treballar amb Java 8, així que cal utilitzar aquesta versió. En principi com ja hem comentat més amunt, és suficient una versió JRE de Java 8, però en el nostre cas, com també necessitem desenvolupar certs components, utilitzarem la versió 8 del JDK.

Si el que pretenem és desenvolupar algun component o algun canal per Kevoree necessitem utilitzar Maven, ja que és el framework que utilitza Kevoree per donar suport als desenvolupaments. No podem canviar el directori per defecte de Maven recollit en la variable d'entorn **MAVEN_HOME**, que es C:\Users\<usuari>\.m2, ja que Kevoree no buscarà aquesta variable d'entorn per trobar el directori, sinó que anirà directament a buscar les llibreries al directori per defecte.

Per últim, l'entorn de desenvolupament que hem utilitzat es Eclipse, ja que estem més familiaritzats amb ell per experiència personal i perquè s'integra molt bé amb Maven. Amb tot açò, podem començar a treballar a fons amb Kevoree.

3.3 Estructura bàsica de Kevoree

Per a poder començar a treballar amb Kevoree, l'únic que necessitem és un jar que se'ns proporciona en la pàgina principal de Kevoree. Aquest jar el que fa és alçar l'entorn mínim necessari per treballar en Kevoree. En concret el que s'està fent és alçar una Màquina Virtual **Java** per manejar el desplegament dels models rebuts des dels editors o components locals. Aquesta JVM s'alça sempre sobre el **port 9000**, cosa que ens va portar problemes quan intentàrem arrancar diversos runtimes de Kevoree en la mateixa màquina, cosa que evidentment va ser impossible.

Quan el runtime arranca, se'ns brinda un model bàsic amb sols un node. Aquest node està buit, i està representant, bàsicament, la nostra màquina. El concepte de node en Kevoree queda un poc difús, ja que en veritat en una màquina poden residir diversos nodes. Però, pel fet que no funciona massa bé col·locar diversos nodes en una mateixa màquina, nosaltres sentarem la premissa de: **una màquina, un node**.

Aquests nodes seran, per dir-ho d'alguna forma, la part central del nostre model. Seran l'element principal de Kevoree, que contendrà els nostres components, i sobre els quals treballarem per a l'aplicació de les modificacions als nostres models. Aquests nodes podem dir que seran els contenidors dels nostres components.

Passem a parlar dels components. Quan en l'entorn de Kevoree parlem de components estem parlant de dues coses a la vegada. Per una banda, un component vist des de la perspectiva purament funcional és un element que s'executa sobre Kevoree i que realitza una funció determinada (la que nosaltres li hàgem programat). Però per altra banda, aquest mateix component funcional te també la vista de component del model, de forma que tenim el gran avantatge que simplement ampliant qualsevol desenvolupament de la classe base component de Kevoree, tenim la doble vista element funcional, element del model.

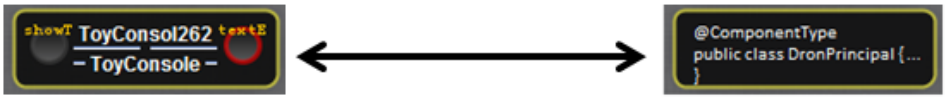

Figura 3.2: Vista de component de model vs vista de component funcional

Hem de tenir en compte que els components de Kevoree sempre van a situar-se dins d'un node. D'ací que els nodes siguen considerats una espècie de contenidors de components.

L'element que queda fora d'aquests nodes són els Canals de comunicació. Els canals de comunicació també són components Kevoree però reben un tractament diferent. Per una banda com ja hem dit, se situaran sempre fora dels nodes. Açò té certa lògica si veiem l'enfocament de Kevoree d'utilització de diversos nodes. Aquests canals pretenen ser una via de comunicació no sols entre diferents components, sinó també entre diferents nodes (d'ací la forta orientació als sistemes distribuïts que té Kevoree). Kevoree ens oferix diverses implementacions d'aquests canals de comunicació, però, per experiència, hem detectat que encara que hauria de ser la part més desenvolupada en un Framework que defén la distribució dels sistemes, és la part més fluixa. Diversos components si, però al cap i a la fi, la funcionalitat real que se'ls pot donar a aquests és discutible, i anem a posar un exemple.

Tenim per exemple el canal SyncBroadcast que és el canal més simple de tots: l'únic que fa és rebre i reenviar missatges. Un simple bus de comunicació.

Per altra banda, tenim el canal RemoteWSChannel. Aquest canal el que ens permet és, teòricament, mitjançant un servidor intermedi, realitzar enviament de missatges a altres components que no necessàriament estiguen en el mateix node. Però tot açò es queda en res a causa de la forma en què està implantat Kevoree (problemàtica de la comunicació entre nodes que afrontarem després). De forma que tenim un excel·lent canal de comunicació que envia els nostres missatges fora del nostre sistema o node però... que sols pots connectar a altre component del mateix model, cosa que el fa inútil.

El que se suposa que permet aquest canal és açò:

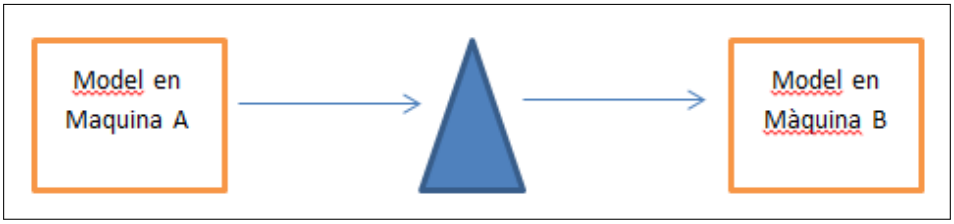

Figura 3.3: Comunicació teorica ideal entre models usant canals Kevoree

Però l'únic que possibilita en veritat fer és açò:

I com aquest canal, diversos. Canals amb asincronia, canals amb retard... però cap útil en la pràctica de la intercomunicació de models.

Amb tot açò, nodes, components i canals podem definir un model en Kevoree. De fet, són els únics elements que anem a gastar per definir un model en Kevoree. Simple però potent. Tenim un lloc on van a funcionar els components, que són

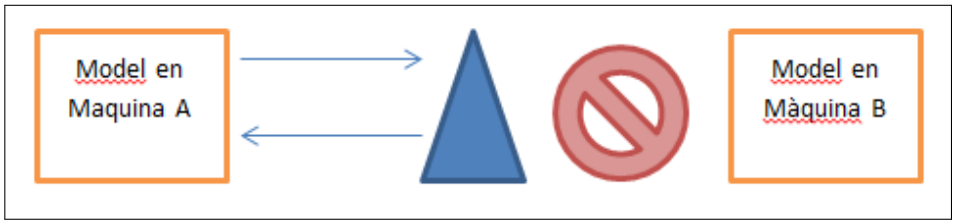

Figura 3.4: Comunicació real entre models usant canals Kevoree

els nodes, tenim els mateixos components, i tenim canals per intercomunicar-los. Simplement hem d'unir-los i començar a treballar.

3.4 L'editor de models de Kevoree

Per poder treballar amb els models, Kevoree incorpora un editor també desenvolupat en Java que ens oferix una sèrie d'operacions que podem realitzar. Quan arranquem l'editor, el que anem a fer és carregar el node que tinguem en execució en el nostre sistema, o en algun sistema remot. Per defecte, el node s'anomenarà node0 i correrà sobre el port 9000. Una vegada arrancat, el model del nostre sistema serà el següent (model bàsic inicial):

L'editor Obtindrà el model actual del runtime Kevoree que tenim arrancat, i el mostrarà. Un cop realitzada aquesta operació, podrem veure el grup de sincronització anomenat "sync"(en verd), un node anomenat "node0", que serà el contenidor dels nostres components, i un enllaç verd que indica que el "node0"és part del grup de sincronització.

Com podem apreciar a la imatge, el nostre "node0"està connectat a un component de tipus WSGroup anomenat sync. Aquest component podem dir que és com el punt principal del nostre model. Aquest s'encarregarà de coordinar tots els nodes que estiguen connectats a ell, a més de poder definir quin port utilitzarà cadascun d'aquests nodes.

Aquest editor ens proporciona diferents formes de modificar el model. La primera i més intuïtiva és mitjançant arrossegar i soltar components. A la part esquerra tenim diferents elements que podem afegir al nostre model, des de nodes, fins components, passant per canals. Els components s'ubicaran sempre dins dels nodes, mentre que els canals estaran fora d'aquests, ja que una de les característiques que li intenta donar Kevoree és que aquests canals puguen servir de comunicació intra-node però també inter-node.

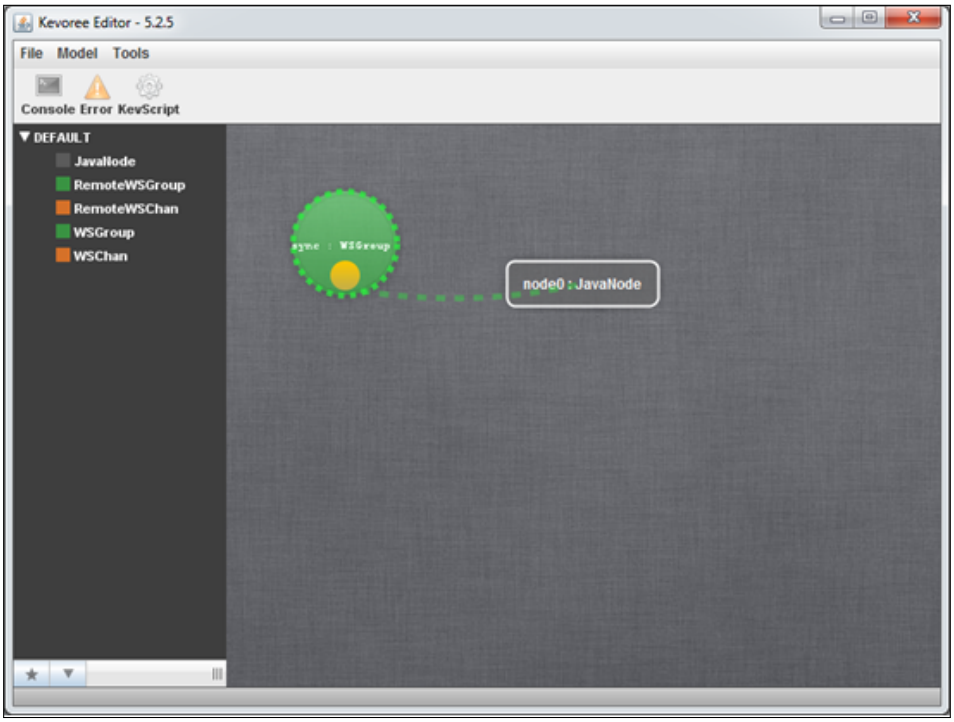

Figura 3.5: Editor de models de Kevoree

Quan provoquem un canvi en el model que afecte un node en concret, el sistema encara no s'ha vist afectat. Açò ens dóna un gran avantatge, que és que podem estar modificant el model mentre el sistema s'executa, i el sistema no es va a adonar de res, seguirà funcionant de forma normal. Ara bé, una vegada els nostres canvis han acabat i volem aplicar aquests canvis al model al sistema, deurem donar el senyal. Açò es fa polsant el botó "push" que incorpora cada node. Quan ho fem, el sistema és redesplega amb els nous canvis que hem introduït al model.

Quan fem el "push", el model passarà a aplicar-se al sistema. A la consola podrem veure si els canvis s'han aplicat correctament o no, ja que per l'eixida principal ens mostrarà alguna cosa similar a aquesta:

En el cas que els canvis no es puguen aplicar correctament, en compte de aparéixer un "End deploy result=true", apareixeria un "End deploy result=false", i ens imprimiria als logs algun tipus d'error.

Figura 3.6: Finestra de propietats d'un node Kevoree

3.5 KevScript

Altra forma de provocar el mateix efecte en el model és mitjançat KevScript. KevScript és el llenguatge de tipus script que defineix Kevoree. L'editor incorpora una consola molt bàsica que és capaç d'interpretar i aplicar les ordres que li donem mitjançant aquest llenguatge.

Imaginem que per exemple, volem afegir un component de tipus DronVigilancia al nostre model, i més concretament el volem situar dins del nostre node0. L'orde en llenguatge KevScript que utilitzaríem seria aquesta:

Estem dient que volem afegir un component de tipus DronVigilancia/1.0-SNAPSHOT (si, hem d'indicar també la versió), que es va a anomenar DronVigilancia1 i que el volem afegir al node0. Una vegada fet açò, s'haurà afegit al model, però encara no estaran reflectits els canvis en el sistema, com en el cas anterior, i també haurem de polsar en el botó "push" que hem anomenat anteriorment.

KevScript es un llenguatge bastant simple. Compta amb unes ordres bàsiques:

- **include**: per importar llibreries.
- **add**: per afegir algun element al nostre model.

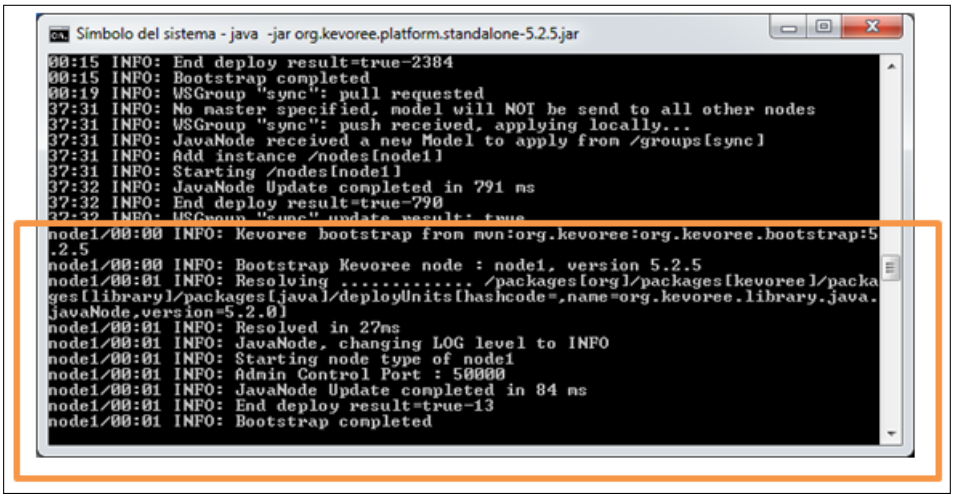

Figura 3.7: Resultat per log de programa d'un canvi de model correcte

```
add node0.DronVigilancia1 : DronVigilancia/1.0-SNAPSHOT

start node0.DronVigilancia1
```

Figura 3.8: Secció de script en KevScript per afegir un component al node0

- **set**: per modificar alguna propietat de algun dels elements del model, per exemple un set DronVigilancia.id = 'Dron1' serviria per modificar el identificador del nostre Dron, que es un camp prèviament definit en el component.
- **bind**: serveix per enllaçar components amb canals de comunicació

Aquestes ordres, combinades amb els diferents elements que ja hem comentat, poden definir tot el model sencer. De fet, un element molt interessant, encara que no l'anem a emprar en aquest treball, és el document "kevs" que porten incorporat tots els components Kevore. Aquest document el podem afegir dins del nostre projecte Maven, de forma que per arrancar un runtime Kevoree, el que farem realment serà arrancar este projecte, que s'encarregarà d'alçar el projecte en un runtime al sistema i posar el model que hem definit en el fitxer kevs en marxa.

A continuació, comentarem com Kevoree enfoca i posa en pràctica les tècniques de models en temps d'execució.

3.6 Kevore i tècniques de Model@Runtime

Com ja hem comentat, Kevoree és un Framework que ens permet vàries coses. Però la principal i per la qual ens hem decantat per utilitzar i investigar més sobre aquesta plataforma és perquè ens permet de forma fàcil l'aplicació de tècniques de modelatge en temps d'execució.

Com bé sabem, les tècniques de modelatge en temps d'execució ens van a permetre diferents coses. La primera, bàsicament, és modelar el sistema. És a dir, tindre una representació fidel del sistema a escala alta i amb un gran nivell d'abstracció: no tenim per què saber com està dissenyat per dins cadascun dels components, sinó que ens limitarem a dissenyar una arquitectura del sistema via model. Per açò, Kevoree utilitza una representació pròpia del sistema, amb la seua pròpia representació dels components que anem a utilitzar.

Com ja hem comentat, no hi ha documentació clara al respecte, però, bàsicament l'estructura que segueix Kevoree per fer possible açò podríem dir que és de la següent manera:

- Kevoree definix un meta model que inclou tots els elements de modelatge que anirem a utilitzar. És a dir, aquest metamodel descriu l'estructura del model que podrem definir. La base del metamodel és el Cloud, el núvol, ja que Kevoree està enfocat a sistemes distribuïts.
- A partir d'aquest meta model, definirem el nostre model, utilitzant els elements que ens proporciona aquest.
- Mitjançant l'aplicació de tècniques de transformació, Kevoree s'encarrega de transformar aquest model en un sistema funcional.

Com qualsevol entorn definit via models, tenim el gran avantatge de poder modificar aquest model sense haver d'accedir a la part de codi del sistema. I com ja hem dit, a més a més, Kevoree ens oferix una transformació ràpida i eficaç de model a codi, de forma que podem modificar per complet el model, encara que el sistema s'estiga executant, i aplicar els canvis per transformar aquest sistema directament en temps d'execució.

És a dir, per concloure, Kevoree ens oferirà un metamodel que és, la base de modelatge que anem a poder utilitzar, a més d'elements ja definits que podran formar part d'aquest model. Nosaltres definirem models que complisquen les restriccions d'aquest meta model i, a més a més, podem afegir nous elements dissenyats per

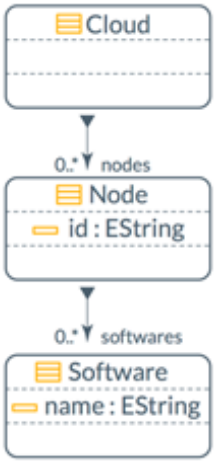

Figura 3.9: Estructura bàsica de Kevoree.

nosaltres sempre que complisquen les restriccions o bases que planteja el meta model. I com ja hem comentat, l'editor de Kevoree serà el que ens permetrà definir de forma gràfica eixe model (també comptem amb KevScript, per definir-lo via codi).

Per realitzar aquestes modificacions de model temps d'execució, Kevoree segueix uns pasos molt marcats. Primerament, hem de realitzar la modificació del model, o bé directament moguent els components que formen el model, afegint, llevant, o modificant aquests, o bé amb KevScript, o bé via programari, de forma que podem automatitzar-ho per què siga el nostre propi component el que realitze aquest canvi de model.

Però el model que modificaren no serà exactament el model que tenim en execució sinó que, Kevoree, ens dóna la possibilitat de fer una còpia d'aquest model, aplicar-li els canvis i, posteriorment, aplicar este nou model per substituir el model que actualment està en funcionament.

Açò ens planteja alguns dubtes. Per exemple, en cas que entre que clonem el model i l'apliquem és produeixa algun canvi al model que està corrent, podria donar-se una situació en la qual eixe canvi fóra sobreescrit per una modificació d'un model anterior al que estava corrent quan s'ha produït el canvi. Pel que hem treballat amb Kevoree, ho veiem una situació una mica complicada però, que veient com funciona aquest, no podríem descartar i, en cas de sistemes més crítics, hauríem

de tindre alguna forma de bloquejar la modificació del model mentre algú ja l'està modificant.

En el següent apartat comentarem com es realitzen els desenvolupaments de component en Kevoree, part important del nostre projecte, ja que, per poder fer que els nostres components siguen autoadaptatius, cal donar-los la funcionalitat a través del codi font d'aquest.

3.7 Desenvolupaments en Kevoree

Per a desenvolupar qualsevol component o canal amb Kevoree, utilitzarem, en el nostre cas i com ja hem comentat abans, Maven i Eclipse. Maven és una eina de programari per a la gestió i construcció de projectes Java. Maven utilitza un Project Object Model (POM) per descriure el projecte de programari a construir, les seues dependències d'altres mòduls i components externs, i l'ordre de construcció dels elements. Ve amb objectius preestablerts per a realitzar certes tasques clarament definides, com la compilació del codi i l'empaquetament.

Aquest és un dels grans avantatges que ens ofereix Maven. Al nostre projecte que després descriurem, ens val seguir l'estructura que utilitza Kevoree com a base per als seus projectes (molt semblant a l'estructura base d'una web-app de Maven) i afegir les dependències corresponents al POM. Amb açò, Maven s'encarregava de descarregar i enllaçar i eixes dependències i de generar correctament el fitxer .jar que posteriorment serà el nostre component.

Per altra banda utilitzarem Eclipse com a IDE per realitzar els desenvolupaments de Kevoree, que al final seran desenvolupaments de classes Java amb les particulars anotacions i elements que incorpora Kevoree.

Com a punt de partida, començarem important un projecte base de Kevoree, que té l'estructura que podem veure a continuació:

Es a dir, un projecte sobre Kevoree compta amb els següents components:

- **pom.xml**: Fitxer de configuració Maven, amb tots els plugins i dependències necessàries per compilar i fer funcionar el projecte
- **kevs** > **main.kevs**: Aquest fitxer és opcional, però en cas d'usar-lo inclou un script en format KevScript. Aquest script servix per arrancar el projecte, de forma que ací podem definir des de propietats inicials que deu tindre el nostre component, fins a una configuració completa d'un model. De fet, un projecte amb aquest format, però sense cap tipus de funcionalitat, sols amb aquest fitxer, es pot fer servir perfectament per definir un model dins d'un runtime Kevoree. En cas de voler fer açò i amb el script ja definit dins

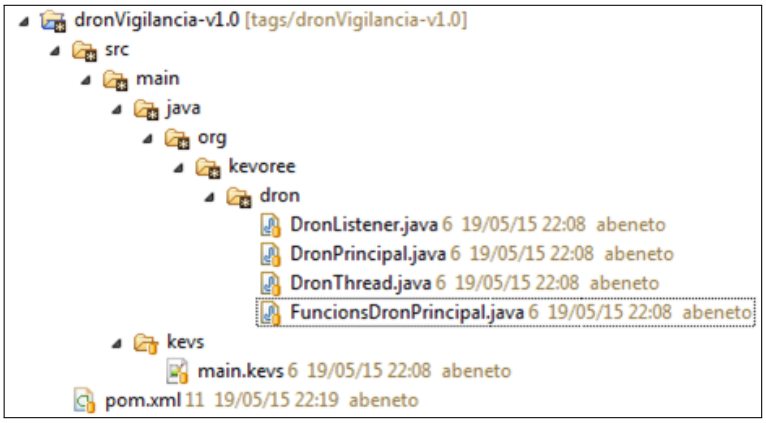

Figura 3.10: Estructura bàsica d'un projecte Maven per a Kevoree

del fitxer, en compte d'arrancar un runtime Kevoree de zero i pegar aquest script en l'editor, podem arrancar directament el fitxer main.kevs (sempre dins d'un projecte Maven) amb la següent ordre:

```
mvn kev:run
```

- **src**: Ací inclourem les classes Java que van a donar funcionalitat al nostre component. En la secció en la que expliquem com hem desenvolupat el nostre projecte entrarem en detall a explicar cadascuna de les classes.

En cas de que no optem per arrancar el runtime Kevoree des de un fitxer main.kevs, compilarem el projecte amb l'orde Maven:

```
clean install.
```

Amb açò, com ja hem comentat abans, se'ns generarà un fitxer .jar, que posteriorment podrem importar al nostre runtime des de l'editor de Kevoree accedint a la secció del menú superior **Model > Load Library**

Capítol 4

Cas d'estudi. Modelatge d'un sistema ubic en l'entorn d'una ciutat intel·ligent.

L'anomenat Internet de les coses (IoT), que és un concepte referit a la interconnexió digital d'objectes quotidians a Internet, és un concepte que cada vegada està més present en la nostra societat. La gent tendeix cada vegada menys a interactuar de forma directa amb els dispositius que ens rodegen, per avançar cap a un model de societat en la qual aquests elements interconnectats seran capaços de comunicar-se entre ells i ser més actius cap a nosaltres que nosaltres cap a ells. Aquests elements són capaços a més d'autoadaptar-se i actuar en funció del context en què es troben, les circumstàncies o l'ambient. Amb tot aquest entramat funcionant, la intel·ligència ambiental o ubiqüitat és col·loca al capdavant d'un nou model de societat.

La intel·ligència ambiental proposa la creació d'entorns intel·ligents, entorns que són capaços d'autoadaptar-se per cobrir gustos i necessitats de les persones. És, ara com ara, el vessant més "humà" o més "al servei de la persona" de forma directa que podem trobar a l'àmbit de la computació. El que intentem modelar amb Kevoree, i que seguidament explicarem, és un entorn que podríem considerar com un subsistema d'una ciutat intel·ligent.

A una ciutat intel·ligent tindríem molts sistemes que interactuaren per dotar-la d'aquesta "intel·ligencia". Podríem tenir sistemes encarregats de la generació i distribució de l'energia, sistemes capaços de gestionar el transport públic adaptant-se a les necessitats de les persones que la formen... I d'altra banda també tindríem un sistema que gestionaria per exemple la seguretat i el benestar dels seus habi-

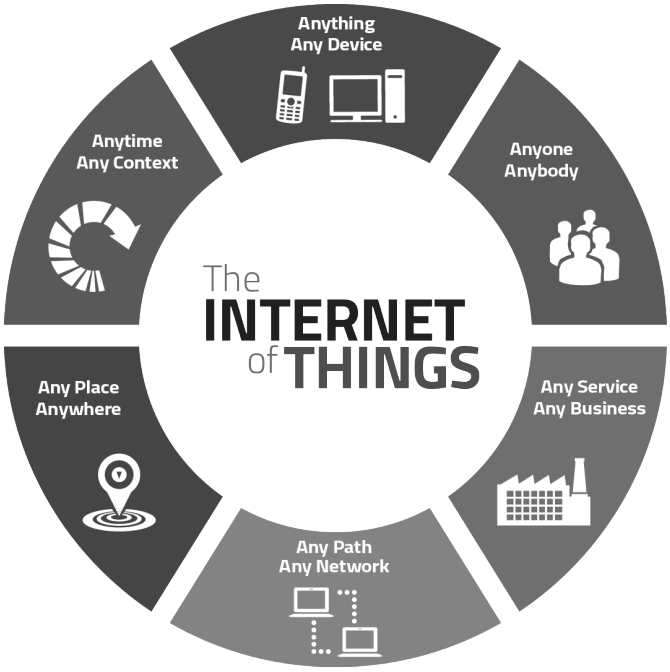

Figura 4.1: Gràfic representatiu de què és IoT. Font: AGT International

tants. Dins d'aquest sistema pensarem a desenvolupar un sistema de vigilància centrat en el tràfic. Un sistema format per drons que seran capaços d'actuar per ells mateixos. Aquests drons vigilaran tot el sistema de tràfic de la nostra societat fictícia i seran capaços de coordinar-se amb la xarxa global de semàfors que forma la nostra ciutat (evidentment, semàfors intel·ligents interconnectats) si qualsevol cosa ho requereix.

Passarem en el següent punt a definir de forma més concreta el nostre prototip, exposarem quina proposta hem seguit per definir-lo i com l'hem desenvolupat. A més a més, parlarem de tots els problemes que hi han sorgit durant el treball i la investigació de Kevoree, com els hem pogut resoldre, o com no els hem pogut resoldre i hem hagut de readaptar el nostre plantejament. Per acabar, exposarem una xicoteta conclusió dels avantatges, desavantatges, problemes i sorpreses que ens hem trobat treballant en Kevoree, fins on creguem que podria arribar Kevoree, i plantejarem propostes per possibles treballs futurs.

4.1 Plantejament inicial

Començarem definint l'àmbit on es mourà el desenvolupament del nostre prototip: Les ciutats intel·ligents. Les ciutats intel·ligents comencen a ser una realitat en alguns punts del planeta, però és una cosa que encara té molt a fer i que suposa una extrema minoria respecte al que segueixen sent les nostres ciutats. És per això que no existeix un, per dir-ho d'alguna forma, patró o model a seguir per plantejar el que hauria de ser una ciutat intel·ligent. Però nosaltres considerem que una ciutat intel·ligent és un gran sistema que engloba subsistemes més menuts capaços d'interactuar entre ells per formar un gran tot que funciona amb perfecta sincronia i dota a la ciutat d'aquesta espècie d'intel·ligència. Si ens fixem en la descripció que apareix, en Wikipedia, ens adonarem que açò no sols afecta l'àmbit més purament tecnològic o computacional, sinó que afecta a tots els àmbits que conformen la societat:

> *"es refereix a un tipus de desenvolupament urbà basat en la sostenibilitat que és capaç de respondre adequadament a les necessitats bàsiques d'institucions, empreses, i dels mateixos habitants, tant en el pla econòmic, com en els aspectes operatius, socials i ambientals"*

Tenint en compte açò, en el nostre projecte ens centrarem més en l'àmbit operatiu, de seguretat i control d'aspectes com el transit o incendis a una ciutat.

El plantejament que ens proposem és definir un prototip d'un sistema que fóra capaç de, mitjançant drons, controlar les principals vies de comunicació de la nostra ciutat intel·ligent, de forma que els nostres drons estaran atents per si es produeix un accident, algú comet una infracció o si, per exemple, hi ha algun incendi prop d'alguna d'aquestes vies de comunicació que impossibilite la correcta circulació del transit. Els nostres drons (com a sistema) interactuaran amb els semàfors de la nostra ciutat. Aquests semàfors també poden interactuar entre ells, de forma que poden passar de simplement canviar de roig a verd per deixar passar o no el transit, a donar senyals lluminoses d'alerta per desviar el trànsit. Açò implica que els nostres components seran capaços d'intercomunicar-se, auto adaptar-se i actuar de forma que el ciutadà que utilitza eixes vies de comunicació no tinga constància de tot l'entramat de sistema que té darrere ni que haja d'interactuar de forma explícita amb el sistema per obtenir un servei per part d'aquest. És a dir, tindrem un sistema ubic, capaç d'intercomunicar-se i auto adaptar-se a les condicions canviants de l'entorn.

Hem de recordar que el que anem a dissenyar és un prototip, ja que creguem que Kevoree fins al moment no està cent per cent madur per fer una implementació real, de forma que els nostres components inclouen una alta càrrega de simulació (evidentment, per simular els esdeveniments als quals haurien de respondre en una situació real). Però creguem que a la velocitat que avança Kevoree, i sobretot si es

resolen els problemes amb els quals encara conten de comunicació entre models i de sincronització entre models, a curt termini podria ser una solució perfectament vàlida per a implementar un sistema real, amb el gran avantatge del treball amb models.

4.2 Elements que formaran el sistema

Per a la implantació del sistema utilitzarem un enfocament orientat a models en temps d'execució, i per això usarem Kevoree.

El nostre sistema comptarà amb els següents elements: Per una banda, tindrem un runtime Kevoree arrancat que farà de base per arrancar el nostre sistema. A més, comptarem amb diversos elements que hem desenvolupat per donar-li vida al nostre prototip:

- **Dron Principal**: Tindrem un component que farà de simulació d'un component dron amb les funcions de coordinació i centralització dels diferents drons del sistema. Aquest dron serà l'encarregat de rebre les comunicacions dels diferents drons i en cas de ser necessari, comunicar-se amb l'element principal de l'altre subsistema

- **Dron Tràfic**: Aquest component que farà de simulació d'un component dron que serà l'encarregat de vigilar les infraccions de tràfic. Enviarà un avís al dron principal cada vegada que es produïsca una infracció perquè aquest actue en conseqüència i el dron de tràfic puga seguir controlant.

- **Dron Vigilància**: Aquest component que farà de simulació d'un component dron que serà l'encarregat de vigilar les principals vies de comunicació. Formarà part del mateix subsistema del dron de Tràfic.

- **Coordinador semàfor**: Aquest component que farà de simulació d'un component que serà l'encarregat de coordinar els diferents semàfors que formen el sistema. A més, podrà rebre comunicacions del dron principal en cas que algun dels drons li comunique alguna incidència que implique canviar la funcionalitat o estat d'algun dels semàfors que formaran el sistema.

- **Semàfor**: Aquest component que farà de simulació d'un component de tipus semàfor, que s'encarregarà bàsicament de dirigir el trànsit, però també podrà canviar el seu comportament habitual de canviar de roig a ver i ver a roig, en cas que hi haja alguna comunicació del coordinador de semàfors que així li ho indique.

- **DronUtils**: Aquest component no tindrà representació en el model del nostre sistema, ja que solament és una llibreria que incorporarà el tipus de missatges que poden enviar i rebre els altres elements i la forma de llegir-

los i crear-los, a més d'incorporar certes constants comunes a tots aquests components.

Una vegada definits els components que formaran part del nostre model, haurem de veure quins altres elements que ens pot proporcionar Kevoree ens poden ser útils per conformar el nostre model. Per una banda, no anem a fer cap modificació als Nodes que ofereix Kevoree, ja que la seua funcionalitat és suficient per al model que volem implantar. Per altra banda, tampoc anem a fer cap modificació en els elements de tipus Channel, però sí que comentarem posteriorment les seues limitacions, cosa que ens ha impedit desenvolupar el sistema exactament com volíem.

A continuació, comentarem com estructurar tots aquests components dins d'un model Kevoree.

4.3 Arquitectura del sistema

Perquè un model i/o sistema funcione de la millor manera possible ha de comptar amb una bona estructura. Per a desenvolupar el nostre prototip tinguérem diverses idees. Però per poder plantejar correctament l'arquitectura del nostre sistema havíem de tindre en compte com anàvem a agrupar els diferents components, ja que teníem tant drons com semàfor, i hi havia diferents formes de relacionarlos. Plantejarem seguidament quin va ser el plantejament teòric de l'estructura del nostre sistema i posteriorment com vam passar d'aquesta arquitectura teòrica a un modelatge ja definit amb Kevoree.

4.3.1 Visió teòrica de l'arquitectura del sistema

Per començar a definir la que posteriorment seria l'arquitectura del sistema però sobretot, el model que anava a representar el nostre sistema, plantejarem una sèrie d'aproximacions teòriques sobre com hauria de ser el nostre sistema. La primera visió que se'ns va ocórrer fou la que podem veure en la imatge 4.2.

En aquest plantejament de sistema tenim els diferents elements que el formen i que ja em anomenat anteriorment distribuïts dins d'un únic sistema o entorn. Aquests elements compten amb la capacitat de comunicar-se tots entre si i formen part del mateix sistema o entorn.

Aquest plantejament en forma de graf pareix simple i fàcil d'implementar via models, però té unes contrapartides que el fan inviable:

- No necessitem una comunicació tots a tots, a més amb Kevoree aquesta comunicació tots a tots no es senzilla d'implementar.

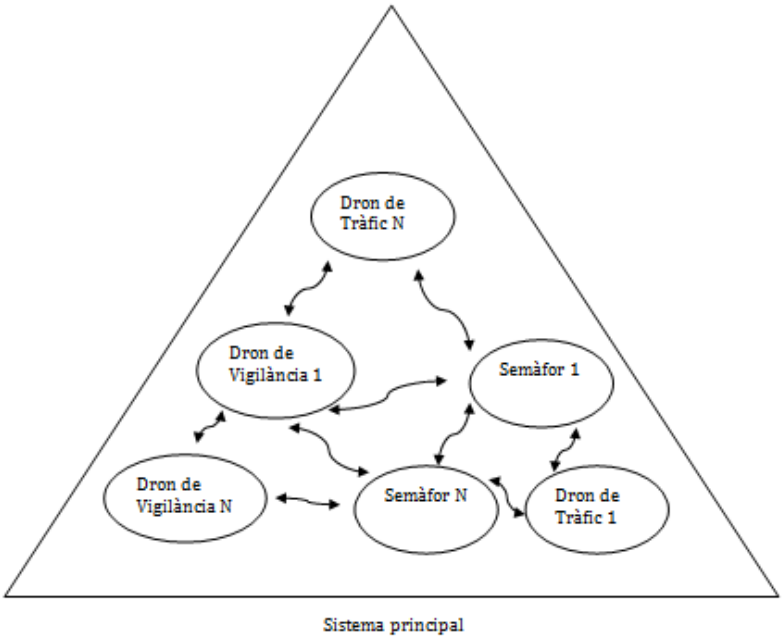

Figura 4.2: Model teòric del sistema. Primera aproximació.

- Amb aquest plantejament, qualsevol element ha de tindre la capacitat de produir un canvi al model, cosa que no ens interessa, ja que preferim centralitzar aquests canvis.
- Imaginem un escenari de fallada d'un component, per exemple, del component Dron de Vigilància 1. Quan açò passe, ha d'haver-hi un element que estiga a l'aguait d'açò i s'encarregue de fer els canvis necessaris en el model per resoldre aquest contratemps. És a dir, el component N hauria de vigilar al component ú, el dos al ú, el tres al dos... I així fins a completar el cicle de N components. Açò és costós de mantenir i difícil d'implementar amb Kevoree.
- Comptem amb un sistema únic, quan en veritat, el que tenim son dos sistemes mesclats: un de drons, i un de semàfors.

Per totes aquestes raons, començarem a modelar una segona aproximació del nostre sistema. Agafant el últim dels punts anomenats anteriorment, establirem un nou sistema. Aquest nou sistema comptaria amb dos subsistemes, el de drons i el de semàfors. Dintre d'aquests subsistemes, l'organització seria semblant a la del primer sistema plantejat. (Figura 4.3)

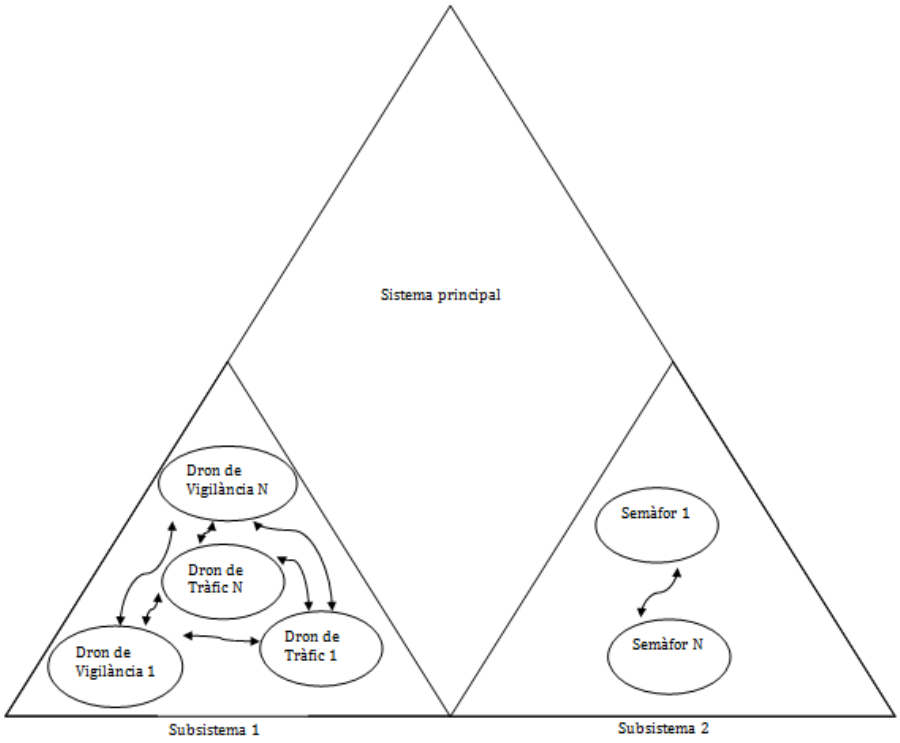

Figura 4.3: Model teòric del sistema. Segona aproximació.

Ara ja tenim dos subsistemes independents. Però clar, seguim comptant amb els mateixos problemes que als punts u, dos i tres del model anterior. A més, es suma un nou problema, la intercomunicació entre els dos sistemes. És evident que per poder col·laborar entre ells s'han de poder comunicar, però, qui començarà aquesta comunicació? Serà bidireccional? Tots els elements han de ser capaços de comunicar-se amb elements de l'altre sistema? Per respondre a totes aquestes preguntes i esclarir-les de la forma més simple i adequada dintre de les possibilitats

que ens ofereix Kevoree, perfilarem un poc més el sistema i arribarem a aquesta aproximació. (Figura 4.4)

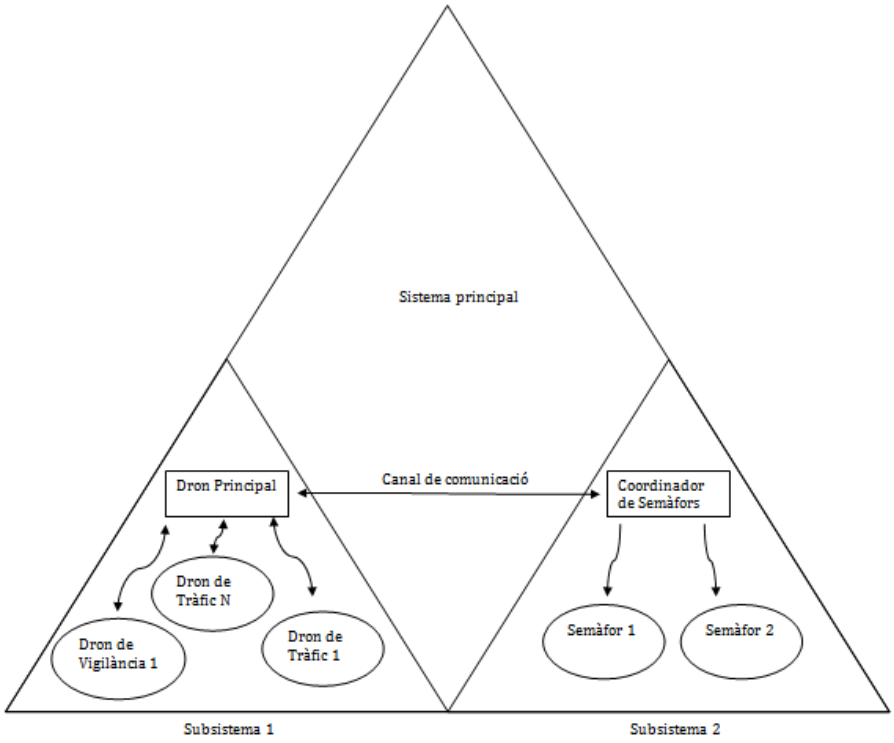

Figura 4.4: Model teòric del sistema. Tercera aproximació.

En aquesta aproximació podem veure que els nostres subsistemes s'engloben dins d'un sistema principal (en aquest cas és l'entorn d'execució de Kevoree, en el cas del prototip AmI que ens ocupa seria la ciutat intel·ligent en si mateixa) i comptem amb dos subsistemes, igual que a l'aproximació anterior. Però aquests subsistemes tenen grans diferències respecte als de l'aproximació anterior.

Per una banda, el subsistema dels drons deixa d'establir una comunicació tots a tots i apareix un nou personatge, el Dron Principal. La comunicació de tots els drons passarà sempre pel dron principal, que farà d'intermediari de missatges i de cap del sistema. De forma que:

- Tenim un cap al sistema que s'encarregarà de centralitzar els diferents processos.

- La comunicació passarà sempre pel dron principal, que s'encarregarà de redirigir els missatges al dron que corresponga o actuarà en conseqüència, per exemple, davant d'un missatge de fallada d'algun dron.
- El dron principal serà l'encarregat de produir els canvis al model, de forma que els diferents drons ja no han de vigilar-se entre si de forma circular, sinó que serà el mateix dron principal el que s'encarregue d'açò.
- El dron principal serà l'encarregat també d'establir la comunicació amb l'altre subsistema.

L'altre subsistema, el dels semàfors, funcionarà exactament igual que el sistema dels drons. És a dir, ara tenim un cap visible de cada subsistema que s'encarrega de centralitzar la funcionalitat i la comunicació entre elements del subsistema i entre els mateixos subsistemes.

Ara bé, que passarà si el cap d'un dels nostres subsistemes entra en condició de fallada? El nostre sistema no contempla una resposta eficaç davant d'açò, ja que és certament costós d'implementar, però seria una condició indispensable si aquest sistema s'implementara en dispositius reals. Una de les solucions que se'ns ocorre és que els caps dels sistemes es suporten entre ells, detectant condicions d'error dels altres caps i resolent-les. El problema d'aquesta solució és que tornem al procés cíclic anterior en què N controla a N-1 i successivament.

Altra de les solucions que se'ns ocorre és que el component puga llançar algun tipus d'esdeveniment i que puga ser interpretat per, o qualsevol component principal d'algun dels subsistemes o, per algun altre component que es dedique exclusivament a la vigilància dels elements principals dels subsistemes. Per falta de nocions més avançades de Kevoree no ens ha sigut possible implementar aquesta solució i la plantegem com un bon punt de partida per algun treball futur amb Kevoree.

4.3.2 Definició pràctica del model amb Kevoree.

Ja hem establit les bases de com volem que siga el nostre sistema. Ara necessitem modelar aquest sistema amb Kevoree per poder obtenir l'objectiu final, que és obtenir un model alterable en temps d'execució que represente fidelment el sistema. I no sols que el represente, sinó que directament pose en funcionament aquest sistema, és a dir, realitze una transformació directa de model a codi en execució.

Plantejarem primer com definir el primer dels plantejaments que hem anomenat anteriorment, i veurem més raons per les quals és inviable i, seguidament, mostrarem el plantejament final del model que hem seguit, d'acord amb l'ultim dels plantejaments que hem mencionat en l'apartat anterior, que serà el sistema final que modelem amb Kevoree.

- **Tipus de sistema tots-a-tots.**

  En aquest tipus de sistema, asociat al primer plantejament definit en l'apartat anterior (Figura 4.2) plantejarem la implantació d'un sòl model en el qual semàfors i drons convivien i s'intercomunicaren entre ells. Cada element del model estaria ubicat dins d'un node Kevoree, simulant un punt individual del sistema. Aquests elements del model s'intercomunicarien amb Canals que proporcionaria el mateix Kevoree. L'estructura que seguiria aleshores el nostre model seria una cosa així:

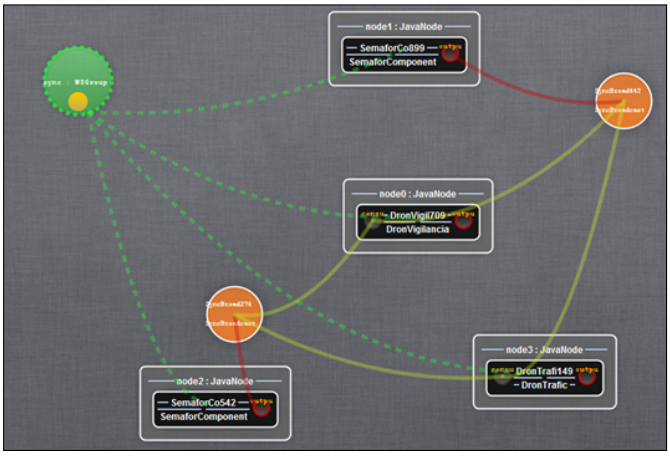

Figura 4.5: Sistema distribuït modelat amb Kevoree multi-node

Avantatges d'aquest plantejament:

- Sistema distribuït total: cada element està en un node, que representa un punt computacional diferent.
- Comunicació directa punt a punt, de forma que ens estalviem passar per un coordinador o punt central.
- Subsistemes integrats: obtenim una espècie d'homogeneïtat, és a dir, no tenim dos subsistemes marcats, un de drons i un de semàfors, sinó que tots formen part del mateix sistema.

Desavantatges d'aquest plantejament (a més dels ja comentats en l'apartat anterior):

- Distribució impossible: No hem arribat al fons de com treballa Kevoree amb els nodes, però ens ha sigut impossible alçar diversos nodes en la mateixa màquina. A més a més, si ho intentem fer amb diverses màquines, (1 node 1 màquina) sí que ho podem alçar, però apareix el problema de la comunicació i coordinació entre nodes (secció següent)

- No tenim elements principals del subsistema que s'encarreguen de comunicar i coordinar, de forma que s'ha d'encarregar de fer-ho el mateix element que detecta l'esdeveniment, deixant de banda la seua tasca principal.

- De la mateixa manera, si algun dels elements cau, els altres haurien d'estar tots alerta per a poder reposar-lo, ja que és un sistema en forma de graf, i no sabem si prèviament el designat a vigilar ja ha caigut, i així successivament. Com que no hi ha un cap visible del sistema, tots deuen estar alerta o idear algun algoritme de control 1 a 1.

- **Tipus de sistema amb nodes niats**

  Altre dels plantejaments que se'ns ocorregué és una cosa prou similar, però en la que drons i semàfors estan agrupats cadascú en el seu node. Açò té exactament els mateixos avantatges i desavantatges que el cas anterior, amb la diferencia del fet que si tenim dos subsistemes més marcats al estar englobats cadascun d'ells en un node diferent. Però el plantejament que vam veure més viable és el següent:

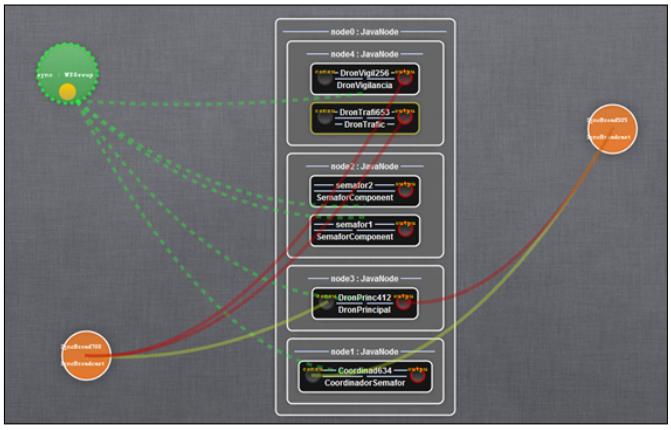

Figura 4.6: Sistema modelat amb Kevoree amb nodes anidats i un de principal

En aquest plantejament el que tenim és un node principal, el node0, que serà com el contenidor principal de tot el sistema, que contendrà altres nodes que seran els nostres subsistemes. Per posar un exemple senzill, el node0 serà el sistema Ciutat Intel·ligent, el node 1 serà el node on estarà ubicat el coordinador dels nostres semàfors, el node 2 serà el node on estaran ubicats els nostres semàfors, el node 3 serà on estarà ubicat el nostre dron principal i el node 4 serà on estaran ubicats els nostres drons.

Avantatges d'aquest plantejament

- El sistema no és realment distribuït, és a dir, el sistema no està en diferents punts computacionals físics, sinó que tot està corrent en la mateixa màquina, però al treballar amb nodes niats, podem simular aquest comportament distribuït, demostrant que la comunicació entre nodes és possible almenys, dins del mateix model.

- Tenim una separació més marcada dels subsistemes, inclús dels components principals d'aquests, de forma que si alguna cosa passa a algun dels nodes, no cau el sistema sencer i és molt més senzill detectar el problema i solucionar-lo.

- Tenim caps dels sistemes, és a dir, tenim un dron principal que s'encarregarà de les comunicacions entre els nodes i la sincronització amb l'altre subsistema, cosa que ja no haurà de fer cap altre dron i podrà seguir fent la seua feina sense problema. Si aquest dron cau, sols cal substituir-lo per altre igual i el sistema no es veu alterat, ni cal preocupar-se per qui és el següent que assumirà eixe rol (com en el sistema anterior)

- Tenim la possibilitat d'establir diverses formes de comunicació: per una banda el dron principal rebrà missatges dels drons que estan vigilant via canals de comunicació de Kevoree. Per altra banda els semàfors es comunicaran d'una forma una mica distinta. En compte de què el coordinador de semàfors els envie un missatge per què canvien d'estat, com en el cas dels drons, el coordinador de semàfors directament realitzarà aquest canvi d'estat modificant el model en temps d'execució i aplicant aquest canvi al sistema.

- Podem canviar o afegir qualsevol component en temps real al model sense afectar el correcte funcionament del sistema.

Desavantatges d'aquest sistema:

- Ens haguera agradat que la comunicació haguera sigut real entre dos sistemes distribuïts, de forma que haguérem pogut tindre el subsistema de drons en una màquina i el subsistema de semàfors en altra, però ha sigut pràcticament impossible. Kevoree compta amb mancances pel

que fa a la comunicació entre nodes computacionals reals, i no sols de comunicació, també de propagació de canvis i sincronització de models.

Per tot açò, ens decidírem per aquesta ultima opció. A continuació explicarem per què no hem pogut implantar el model en diferents punts computacionals reals, a causa dels dos problemes o mancances principals que li hem trobat a Kevoree: la comunicació entre nodes i la propagació i actualització de canvis de model entre nodes.

4.4 Comunicació entre components

Quan començarem a plantejar com desenvolupar els nostres components ens sorgí el dubte de com s'anaven a comunicar aquests. La primera opció que se'ns va ocórrer consistia a llançar algun tipus d'esdeveniment des dels drons i amb algun gestor de Kevoree capturar-los per poder tractar-los i aplicar els canvis al model que foren necessaris. Prompte ens adonarem que aquesta aproximació era inviable a causa de la mateixa arquitectura de Kevoree, que no compta ni amb esdeveniments ni amb un gestor simple i intuïtiu per poder realitzar aquestes tasques. La solució que adoptarem fou la següent:

- Establir un mecanisme propi de creació de missatges, amb uns camps molt marcats perquè sempre seguiren la mateixa estructura.
- Com Kevoree sols ens permet enviar, a través dels ports dels components i dels canals de comunicació, missatges en format String, hauríem d'establir un mecanisme de transformació del missatge a String i de String a l'objecte de tipus missatge que ens havíem creat.
- Establir també un mecanisme perquè els receptors pogueren interpretar aquests missatges de forma correcta.

Per a fer tot açò, crearem la classe **DronMessage**. Aquesta classe s'encarregarà d'encapsular tota la informació que enviem a través dels diferents components de Kevoree que hem creat. Per a fer possible açò, la classe compta amb els següents camps:

- **event** (tipus DronEvents): aquest camp indicarà quin tipus d'esdeveniment estem encapsulant en el nostre missatge, és a dir, que ha provocat que s'envie aquest missatge. Aquest camp és de tipus Enum, és a dir, té uns valors fixes i limitats, per facilitar el tractament de dades.
- **rol** (tipus DronRoles): aquest camp indicarà quin és el rol de l'emissor del missatge. És a dir, si el nostre dron de vigilància és el que envia el missatge, en aquest camp s'indicarà que ha sigut un dron de vigilància. Aquest camp

és de tipus Enum, és a dir, té uns valors fixes i limitats, per facilitar el tractament de dades.

- **idDron** (tipus String): aquest camp indicarà la identitat del dron que ha enviat el missatge. Ens serà útil en casos en què tinga'm que aplicar un canvi de model sobre el dron emissor del missatge, per exemple, en cas que ens arribe una alerta d'avaria o bateria baixa del dron.

- **missatge** (tipus String): aquest camp és totalment lliure i servirà per enviar el missatge que vulga'm. Aquest camp no té cap repercussió respecte al receptor del missatge, simplement és un camp per introduir informació complementària a l'esdeveniment que s'ha generat.

Els diferents components que vulguen enviar un missatge sols hauran de crear una instància de la classe DronMissatge, omplir els camps i envair-lo a través del seu port de comunicació. Per enviar-lo executarem una ordre del següent estil, fent ús dels ports i canals de comunicació que ens proporciona Kevoree:

```
1  DronMessage missatge = new DronMessage();
2  missatge.setEvent(org.kevoree.dron.utils.DroneEvents.
       INTERCEPTAT_INFRACTOR_TRAFIC);
3  missatge.setIdDron(String.valueOf(this.getId()));
4  missatge.setRol(DronRoles.CAMERA_TRAFIC);
5  missatge.setMissatge(getInfraccioRandom());
6
7  this.dronListener.helloDron(missatge);
8
9  public void helloDron(DronMessage helloValue) {
10
11         if (this.conexionActiva()) {
12         }
13
14         outputPort.send(helloValue.toString(), new Callback()
               {
15                 public void onSuccess(CallbackResult
                       callbackResult) {
16
17                 }
18
19                 public void onError(Throwable throwable) {
20                         System.out.println("Error de
                               comunicacio en el dron ");
21                 }
22         });
23 }
```

Codi Font 4.1: Fragment de codi per crear i enviar un missatge.

L'acció outputPort.send és la que ens proporciona Kevoree per defecte per enviar els missatges, a través del outputPort que serà un camp del component de tipus Port que ens ofereix Kevoree. La lògica de com s'envien internament els missatges és totalment transparent a nosaltres.

Quan fem el send, si ens fixem, el que fem és enviar el toString del objecte missatge que hem creat (com ja hem dit, el mecanisme de Kevoree sols permet enviar String). El que hem fet és personalitzar aquest toString, de forma que s'envie en un format que es puga recompondre facilment per quan arribe al receptor poder tornar-lo a transformar en un objecte de tipus DronMessage. El format en el que s'enviarà serà un String que seguirà aquesta estructura:

"event:rol:idDron:missatge".

Aquests dos punts seran els delimitadors de camps amb els quals encapsularem i desencapsularem els nostres missatges. Per altra banda, necessitem una forma de rebre aquests missatges i poder recompondre'ls per què el receptor sàpiga interpretar-los. Ací juga un paper importat la funció de la figura 4.2 que podem veure més avall.

```
1  public DronMessage toDronMessage(String missatge) {
2  DronMessage droneMessage = new DronMessage();
3
4  String array[] = missatge.split(":");
5
6  if(array[0].contains(DroneEvents.AVERIA_CAMERA.toString())){
7          droneMessage.setEvent(DroneEvents.AVERIA_CAMERA);
8  } else if (array[0].contains(DroneEvents.AVERIA_HELIX.toString
       ())) {
9          droneMessage.setEvent(DroneEvents.AVERIA_HELIX);
10 } else if (array[0].contains(DroneEvents.BATERIA_BAIXA.
       toString())) {
11         droneMessage.setEvent(DroneEvents.BATERIA_BAIXA);
12 } else if (array[0].contains(DroneEvents.
       INTERCEPTAT_INFRACTOR_TRAFIC.toString())) {
13         droneMessage.setEvent(DroneEvents.
               INTERCEPTAT_INFRACTOR_TRAFIC);
14 } else if (array[0].contains(DroneEvents.
       SOBREPASAT_PUNT_NO_RETORN.toString())) {
15         droneMessage.setEvent(DroneEvents.
               SOBREPASAT_PUNT_NO_RETORN);
16 } else if (array[0].contains(DroneEvents.DETECTAT_INCENDI.
       toString())) {
17         droneMessage.setEvent(DroneEvents.DETECTAT_INCENDI);
18 }
19
20 if(array[1].contains(DronRoles.CAMERA_INCENDIS.toString())) {
```

```
21              droneMessage.setRol(DronRoles.CAMERA_INCENDIS);
22      } else if(array[1].contains(DronRoles.CAMERA_TRAFIC.toString()
            )) {
23              droneMessage.setRol(DronRoles.CAMERA_TRAFIC);
24      } else if(array[1].contains(DronRoles.VIGILANCIA.toString()))
            {
25              droneMessage.setRol(DronRoles.VIGILANCIA);
26      }
27
28      droneMessage.setIdDron(array[2]);
29      droneMessage.setMissatge(array[3]);
30
31      return droneMessage;
32      }
```

Codi Font 4.2: Fragment de codi per crear i enviar un missatge.

Bàsicament el que fem ací és partir el String que ens arriba, separar-lo gràcies als separadors que hem establert, i reconstruir el missatge a un DronMessage altra vegada. Per què fem açò? Doncs perquè posteriorment per tractar sobre eixe missatge ens serà més còmode poder accedir camp a camp que haver de treballar sobre una cadena de caràcters. Així és molt més fàcil, per exemple, consultar quin és el dron que ho ha enviat, o quin rol tenia, o quin és el missatge que volia transmetre, sols fent un get sobre el camp corresponent de l'objecte que acabem de reconstruir.

Amb açò tenim un sistema de comunicació complet. Si afegim més tipus de rols o més tipus d'esdeveniments, sols hauríem d'afegir-ho als Enums corresponents i a l'element que reconstrueix el missatge i ja tindríem el sistema enllestit una altra vegada.

4.5 Modificacions del model. Utilitzant Model@Run.time amb Kevoree per fer el sistema autoadaptable.

I per a què ens van a servir aquests missatges? Bé, bàsicament per poder comunicar al Dron Principal via missatge que alguna cosa ha passat i s'ha de produir un canvi del model, i per consegüent del sistema.

Imaginem per exemple, un cas en el qual el dron de vigilància estiga quedant-se sense bateria. Aquest dron el que farà serà comunicar la seva situació al dron principal mitjançant un missatge del tipus que hem descrit anteriorment i el dron principal prendrà les mesures corresponents, és a dir, provocarà un canvi de model que es plasmarà en un canvi del sistema.

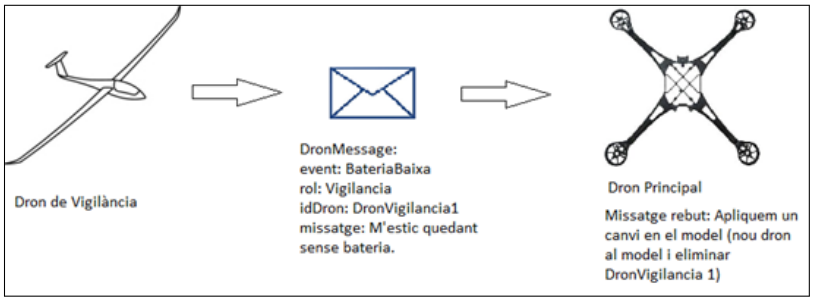

Figura 4.7: Exemple de comunicació entre drons

El dron principal és el que al final aplica un canvi al model, però és el dron de vigilància el que el provoca via comunicació amb missatges del tipus que hem definit.

També es poden provocar canvis de model sense petició expressa de cap element del model, sinó que siga un d'ells el que provoque el canvi. Per exemple, en el cas dels semàfors, és el coordinador el que s'encarrega de canviar el color del semàfor. Per fer-ho el que fa és aplicar un canvi al model, en concret modificar les propietats dels semàfors i canviar-los el color que tenen en eixe moment. Posteriorment aplica el model i el sistema ja compta amb el nou estat d'eixos semàfors.

En qualsevol dels dos casos el que perseguim és modificar el sistema alterant el model en temps d'execució. És més, aconseguim fer-ho sense que nosaltres hàgem d'intervenir, sinó que són els mateixos components els que ho realitzen responent a esdeveniments d'uns o altres components.

Passarem a detallar com funciona Kevoree per alterar el model. Podem fer-ho de diferents formes:

- Modificant el model directament des de l'editor: simplement amb el ratolí, arrossegant components i eliminant-los podem modificar el model. Una volta modificat, estrenyerem el botó "push" del contenidor que volem modificar i els canvis s'aplicaran al model. Aquesta és la forma més intuïtiva de fer-ho però requereix la nostra participació
- Utilitzant el llenguatge KevScript: l'editor de Kevoree incorpora una consola per al llenguatge KevScript que ens permet introduir ordres en aquest format i llançar-les perquè es provoquen canvis en el model. Una vegada s'ha modificat el model, també deurem estrènyer el botó push perquè s'apliquen els canvis al sistema. Aquesta forma és la menys útil, ja que també requereix la nostra participació i, a més a més, requereix coneixements sobre el

llenguatge KevScript, és molt més lenta (tarda molt a executar-se l'escript en la consola de l'editor) i ens fa haver de fer més passes que simplement modificant el model a mà amb l'editor.

- Edició des de els mateixos components: aquesta és la forma que ens interessa per al nostre projecte, perquè ens permet automatitzar la modificació del model, de forma que són els mateixos components els que s'encarreguen de fer eixes modificacions i aplicar-les al model. Aquesta és la millor forma d'aplicar tècniques de modelatge en temps d'execució, ja que prescindim de la nostra participació i provoca que el sistema siga auto adaptable.

Per poder fer açò, hem d'entendre el paradigma que proposa Kevoree i com pot fer açò possible.

Kevoree treballa amb una representació completa del model sobre el qual estem treballant. Aquest objecte, en codi Java, és UUIDModel. Aquest és un objecte de sol lectura, per la qual cosa no podem treballar directament sobre ell. Per resoldre açò Kevoree ens proporciona un "clonador" de models, és a dir, ens ofereix la possibilitat de fer la còpia del model actual per poder modificar-la.

```
1 this.localModel = this.cloner.clone(model.getModel());
```

Codi Font 4.3: Codi per clonar un model a nivell de codi.

Una vegada tenim la còpia del model (localModel), ja podem treballar sobre ella. Per modificar el nostre model clonat, utilitzarem KevScript, però en l'àmbit de codi Java. Ens generarem una cadena de text amb el Script que volem emprar per modificar el model i amb l'ajuda dels serveis que ens ofereix Kevoree l'aplicarem al model clonat. En aquest cas, utilitzarem el servei KevScriptService, que és un servei oferit per Kevoree que s'encarrega d'aplicar un script en KevScript sobre un model que li indiquem (en aquest cas, el clon del nostre model).

```
1 try{
2     kevScriptService.execute(script,localModel);
3 } catch (Exception e1) {
4     e1.printStackTrace();
5 }
```

Codi Font 4.4: Codi per executar els canvis sobre el model.

Una vegada aplicat el script al model, ja tenim la modificació aplicada. Però ara volem també que el nostre model es corresponga al nostre sistema. És per això que necessitem que el nostre model s'aplique, i passe a ser el model del sistema, i no sol un clon de l'anterior model. Per això, Kevoree ens ofereix un altre servei,

anomenat ModelService, que conté una funció Update, a la qual li passarem el nou model modificat i aquest servei s'encarregarà de substituir-lo pel model anterior que és el que actualment està en marxa.

```
this.modelService.update(localModel, new UpdateCallback() {
    public void run(Boolean arg0){
        ;
    }
});
```

Codi Font 4.5: Codi per aplicar els canvis realitzats al model clonat sobre el model actual en funcionament.

Amb aquests tres passos provoquem una modificació del model i l'apliquem al model que actualment està en marxa en el nostre runtime. Açò, juntament combinant amb els missatges i el tractament d'aquests que hem definit ens porta a un nivell d'automatització en el qual el nostre sistema és capaç de respondre a esdeveniments amb canvis de model. És a dir, hem aconseguit un **sistema capaç de modificar el model en temps d'execució**, un sistema **auto adaptable** i un sistema que **és una representació d'un entorn englobat dins de l'àmbit de la intel·ligència ambiental**, un sistema ubic.

Ens falta treballar sobre un punt al nostre sistema, que és la distribució. En un sistema d'aquest tipus real, cada element estarà en un punt computacional diferent, de forma que necessitàrem treballar amb diferents nodes i models i fer-los capaços de rel·lacionar-se entre ells. Però amb Kevoree sorgeixen una sèrie de problemes, que passarem a detallar en el següent punt.

4.6 El problema de la sincronització entre models

Quan ens posem a treballar a Kevoree s'indica que aquest és un Framework fortament orientat a la modelització de sistemes distribuïts, i, si ens parem a analitzar la forma en què està plantejada la modelització dels sistemes té sentit. Kevoree estructura els models amb nodes, que poden estar tots al mateix punt computacional, però la gràcia resideix en distribuir aquests nodes, de forma que podem tindre punts computacionals diferents.

Però, quan intentem modelitzar un sistema d'aquesta manera sorgeix un problema que no havíem tingut fins ara. Imaginem que els nostres dos sistemes, semàfors i drons, van a ubicar-se en dos punts computacionals diferents. Per poder fer açò devíem arrancar un runtime Kevoree en cadascun d'eixos punts computacionals. És a dir, de primeres, comptaríem amb dos models diferents en funcionament en dues màquines diferents.

Açò no es desagrada, a l'inrevés, al tenir dos models podem treballar amb ells de forma separada, i cadascun d'ells modelarà un dels subsistemes que hem plantejat.

El problema apareix quan intentem dues coses:

- Realitzar una comunicació entre els dos models
- Realitzar una propagació de modificacions del model, és a dir, si es modifica el model A i això afecta el model B.

Aquestes dues coses hem sigut incapaços de portar-les a terme, a pesar que Kevoree se suposa que està fortament enfocat als sistemes distribuïts.

Figura 4.8: Vista abstracta de dos models en punts computacionals diferents

Quan intentem modelar dos sistemes en dos punts diferents, tenim una cosa similar al que veiem en la figura 3.9.

Aquests dos models, corrent en maquines diferents, són totalment independents i no tenen cap tipus de comunicació entre ells. Aleshores, açò ens suposa un problema, i posarem d'exemple el nostre cas.

Suposem que en el model A, estem representant el sistema de drons, mentre que en el model B estem representant el sistema de Semàfors. Cadascun d'aquests sistemes compta, com ja hem dit, d'un element principal o cap del sistema, que s'encarregarà de rebre i enviar missatges i aplicar canvis de model segons corresponga.

Però els dos models no tenen cap comunicació, per la qual cosa no poden comunicar-se entre ells. La solució que ens donaven des de Kevoree era la següent: utilitzar un component especial dels canals que ens ofereix Kevoree per intercomunicar els dos sistemes. Això si, amb una peculiaritat: encara que cada model correguera per separat en una màquina diferent, s'havia de fer una fusió (merge) dels dos models per poder tindre una visió d'un model sencer i poder així gastar aquest component canal per intercomunicar-los.

Fins ací la idea té bona pinta. Es tractaria de, per exemple, en el model A, carregar i fer una fusió del model B, de forma que tindríem:

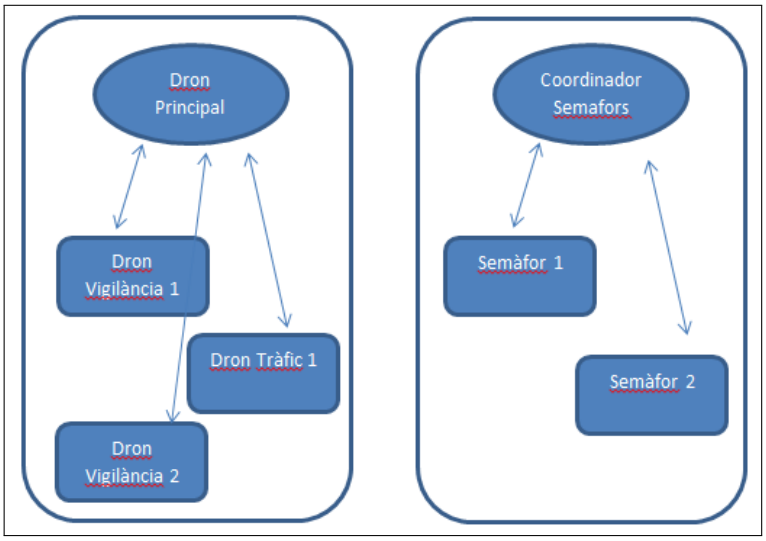

Figura 4.9: Vista dels components que formen els nostres dos models en punts computacionals diferents

- Model A = Model A + Representació del Model B.
- Model B = Model B

La cosa és que fent proves arribarem a comprovar que, si féiem açò el que en veritat teníem era un model que era la representació de tot el sistema, i altre model més menut inalterat.

El problema ve en què, si nosaltres agafem i apliquem el model A (que ara és el model A+B) el que fa no és modificar el model A per una banda i el B per una alta, sinó que el que tenim és que on abans teníem el model A, ara tenim el model A i B (sí, els dos models a l'hora amb la seua implementació física), i el model B no s'hi ha adonat de cap dels canvis.

Ens donarem conter d'aquest funcionament quan, fent la fusió, veiem que en el model que fins ara era el A, també apareixien funcionant els semàfors, que sols haurien d'estar funcionant en el model B.

Per tant, la comunicació entre model A i model B no era possible i per tant, encara era més difícil la propagació si hi havia algun esdeveniment que provocarà algun canvi en algun dels dos models.

Investigant un poc més sobre Kevoree, veiérem que comptava amb un component en les seues llibreries anomenat RestServer. Aquest component el que fa és alçar un servidor RESTful simplement afegint-lo al nostre sistema. Ací veiérem una solució viable, que implicava modificar el component que ens oferia Kevoree per adaptar-lo a les nostres necessitats, fent que aquest servidor rest fera d'intermediari dels missatges entre el model A i el model B. Ací també va ser tasca prou complicada, ja que no veiérem forma fàcil d'estendre aquest component.

Al final, optarem per prescindir de fer un sistema distribuït i ho implementarem directament en el mateix model, cosa que a pesar de tot, no fa que perdem la resta d'avantatges que ens ofereix Kevoree, és a dir, continuem tenint un sistema auto adaptatiu que respon a esdeveniments del mateix sistema modificant els models en temps d'execució.

Capítol 5

Conclusions

El treball amb Kevoree ha sigut una tasca bastant complicada a causa de la gran falta de documentació amb què conta tant l'entorn en si com els components que el formen. Així i tot creguem que és una eina amb molt de potencial si l'emprem de forma correcta.

La forma en la que ens permet modelar un sistema és molt bona. És simple, és intuïtiva, i ens permet modelar un sistema simple en molts pocs segons. A més, la llibreria de components que incorpora és molt extensa i, encara que té moltes coses per millorar, la majoria de components són prou estables per poder gastar-los sense haver de modificar molta cosa del seu codi. També volem destacar la part de KevScript.

Pareix un llenguatge amb bastant potencial dins de l'entorn de Kevoree, que compta amb una integració simple però potent tant en Java com en JavaScript, cosa que li dóna un plus per al desenvolupament de forma assequible de nous components.

D'açò ultim també estem molt satisfets, gràcies al fet que la integració amb Maven facilita molt la tasca de desenvolupar qualsevol component que vulguem. Al final, és suficient amb seguir l'estructura que ens marca Kevoree, utilitzar correctament les anotacions que se'ns donen, i ja tenim un component perfectament funcional per integrar al nostre model. I com el desenvolupament de la funcionalitat interna és purament Java, les possibilitats són infinites.

En el nostre cas, no volíem aprofundir molt en el desenvolupament, sinó que simplement volíem mostrar la possibilitat de realitzar un prototip d'un entorn modelat amb Kevoree en l'àmbit de la intel·ligència ambiental, que fóra capaç d'autoadaptar-se modificant el model que el representa, que ens permetera modificar-lo donat el cas, i que tot açò ho fera seguint un enfocament orientat al treball amb

models en temps d'execució. En aquest aspecte, Kevoree compleix sobradament els objectius que ens havíem plantejat, ja que Kevoree demostra que és capaç, a més de forma prou simple i eficaç, de modificar el model en temps d'execució i, a més, de diferents formes, com ja hem pogut veure en apartats anteriors.

Posar com a nota negativa, això si, la dificultat que suposa la intercomunicació de models i la propagació de canvis entre ells. Ens ha costat molt de temps intentar esbrinar com funciona internament Kevoree en aquest aspecte però no ha hagut forma de poder-ho afegir al nostre model. De fet, la possibilitat de facilitar la comunicació entre models i la propagació de canvis podria donar una espenta a Kevoree dins del món del treball amb sistemes auto adaptatius i que treballen amb models en temps d'execució. Ho veiem suficientment dificultós d'aconseguir fins al moment, de forma que un sistema que, a més de totes les característiques que hem comentat anava a tindre el gran avantatge de ser distribuït, es queda a meitat camí.

Podem establir una llista de punts forts i fluixos de Kevoree:

- Punts possitius:
  - Facilitat de posada en marxa
  - Facilitat d'ús de l'entorn. És intuïtiu i funcional.
  - Gran llibreria de components.
  - Modificació de models de diferents formes: directe sobre el model, via KevScript, via codi
  - Desenvolupament de components senzill
  - Pocs elements per formar un model potent: components, canals, nodes i grups de sincronització.
- Punts negatius:
  - Ens hem trobat algun bug pel fet que és un framework encara en desenvolupament.
  - La utilització del llenguatge KevScript des de l'editor és molt lenta.
  - Les modificacions de model fetes via component (des de codi) no es plasmen directament a l'editor, cal recarregar el model per vore aquests canvis aplicats.

– A pesar de tindre una forta orientació als sistemes distribuïts, ens ha resultat impossible alçar un. És més, en la mateixa documentació de Kevoree no hem trobat cap apartat que explique açò de forma clara

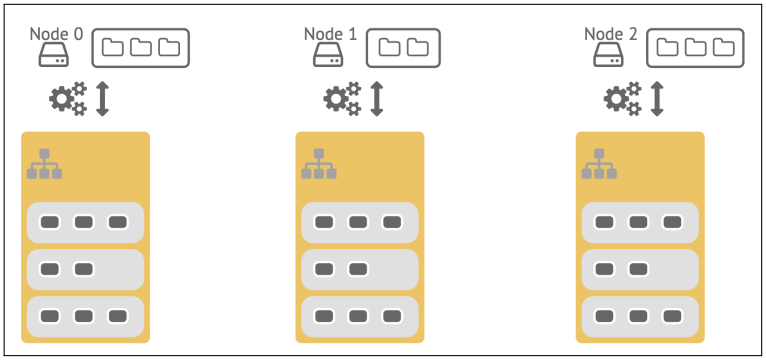

Figura 5.1: Sistema de nodes distribuït modelat amb Kevoree, un dels objectius a aconseguir.

Podem concloure per tant, que Kevoree és una eina en creixement i amb algunes mancances, però amb molt de potencial. Una eina a tindre en compte quan treballem amb models en temps d'execució i que ha demostrat que amb molt poc de desenvolupament, pot treballar de forma satisfactòria amb sistemes que necessiten autogestionar-se.

Pet totes aquestes raons hem escollit Kevoree per plasmar aquest xicotet prototip. Esperem que en un futur no molt llunyà es transforme en una eina de referència per al treball amb models en temps d'execució, i amb sistemes auto adaptatius i distribuïts, de forma simple, eficaç i sobretot molt intuïtiva, que és el gran punt fort de Kevoree.

5.1 Treball futur

Tal volta, la part més interessant i que se'ns ha resistit un poc en aquest treball de fi de màster és el treball amb nodes distribuïts. Creguem que un dels punts molt interessants de Kevoree seria el treball amb la comunicació i la propagació de canvis als models. És per això que creguem que aquest treball és perfectament ampliable.

Com a treball futur seria interessant trobar una forma no sols de comunicar diferents models o nodes de diferents models, sinó que també seria molt interessant obtenir alguna forma de propagació de canvis entre models, de forma que qualsevol

canvi a un model que suposara un canvi a un altre model, es poguera propagar de forma simple utilitzant components que ens oferiria Kevoree. Enfocar el desenvolupament en l'entorn dels canals de Kevoree seria interessant, per poder, arribat el cas d'aconseguir desenvolupar un, col·laborar amb aquest canal de comunicació amb la comunitat que dóna suport a Kevoree i integrar-lo en les seues pròpies llibreries.

Altre aspecte que ens pareix interessant com a treball futur i que ja comentem a la secció 4.3.1, és que els components principals dels nostres subsistemes, o més generalment, qualsevol component, puga llançar algun tipus d'esdeveniment i que puga ser interpretat per altre component en qualsevol punt del sistema de què forma part o de qualsevol altre sistema amb el qual tinga establida una comunicació. Ja hem comentat que per falta de nocions més avançades de Kevoree no ens ha sigut possible implementar aquesta solució, però creguem que, trobada la forma de fer-ho, canviaria molt la forma de plantejar els sistemes i ens donaria una gran versatilitat en aquest i a més, seria un gran avanç per a començar a treballar també amb sistemes tolerants a fallades amb Kevoree.

Altre dels punts sobre els quals seria interessant ampliar aquest treball seria la implementació d'un model sobre uns components reals. Al principi, ens plantejarem la meta de poder implementar el nostre sistema sobre drons reals, cosa que descartarem finalment per falta de temps i per falta de robustesa de part del sistema en si. Creguem que seria un bon camí a seguir quan Kevoree avance un poc i la seua fiabilitat siga més alta i quan els diferents entrebancs pel que fa a comunicació i propagació de canvis de model s'hagen resolt.

Apèndix A

Apèndix

En aquest apartat inclourem tots els elements que considerem importants del treball però que no tenien cabuda en un capítol principal d'aquest, com poden ser fragments de codi font, referencies als repositoris de codi utilitzats i passos per alçar i modelar el nostre sistema, entre d'altres.

A.1 Repositoris i metodologia

Per al desenvolupament del nostre codi font i, d'acord amb seguir unes bones pràctiques de versionat i realització de còpies de seguretat de codi, hem utilitzat un repositori Subversion per als backups del nostre codi i la realització de còpies del nostre codi i alliberació de versions.

Ens agradaria comentar també que hem seguit una metodologia de desenvolupament molt propera a SCRUM, però sense seguir tots els processos associats al cent per cent. Els desenvolupaments que hem realitzat els hem fet basant-se en sprints, tal com marca aquesta metodologia, aconseguint en cadascun d'aquests sprints una versió estable i funcional.

El nostre repositori està estructurat de forma que tenim una carpeta per component, dins de la qual tenim tres carpetes:

- trunk: ací es troba la versió en desenvolupament. La versió trunk sempre s'etiquetarà de la forma *NomDelPaquet-versio-SNAPSHOT*. Per exemple, DronPrincipal-1.1-SNAPSHOT

- branch: ací es troben les versions que son branques dels desenvolupaments en curs o de versions tancades. Seguixen la mateixa nomenclatura que les versions de la qual s'està creant la rama.
- tag: ací es troben les versions finals, és a dir, el resultat final de desenvolupament del trunk en cada sprint. Aquestes versions alliberades seguiran la nomenclatura *NomDelPaquet-versio*, per exemple DronPrincipal-1.1

També hem de tenir en compte que, en cas de què la modificació realitzada al sprint altere en gran mesura la composició del component, es considerarà un canvi major i canviarem el primer nombre indicador de la versió i deixarem a zero el segon. Per exemple, de la versió 1.1-SNAPSHOT a la 2.0-SNAPSHOT.

Aquestes pautes s'indiquen pel fet que el repositori és obert, i tot el món pot accedir per realitzar modificacions al codi.

La ruta d'accés al repositori és la següent:

https://subversion.assembla.com/svn/kevoree-abeneto/

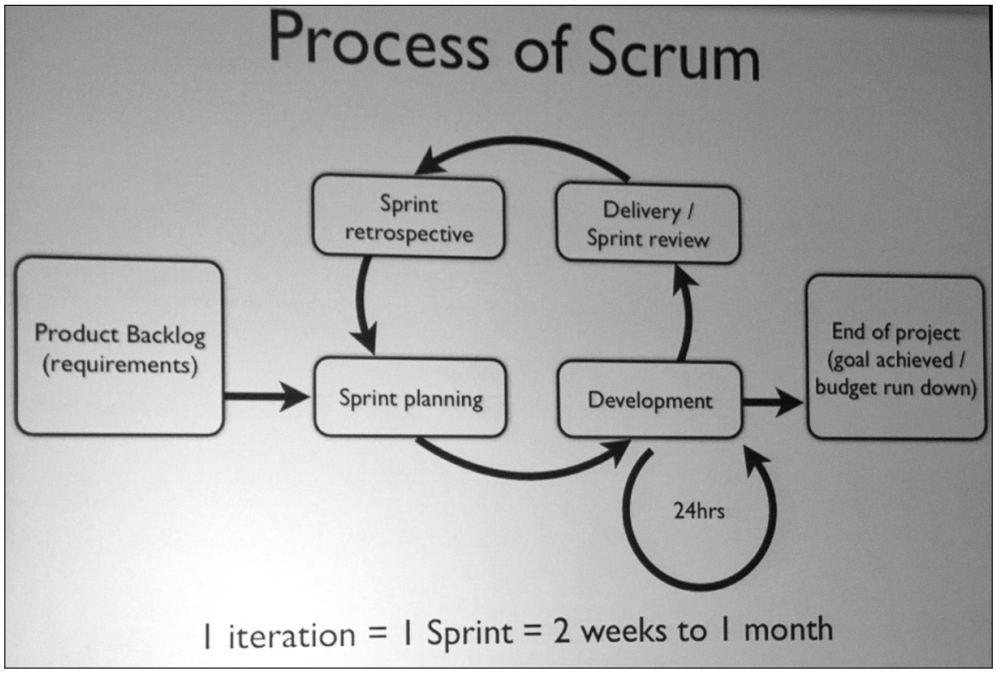

Figura A.1: Procés de transformació de models

A.2 Codi font. Dron Principal

En aquesta secció inclourem fragments de codi font rellevants per al nostre treball, acompanyats dels comentaris necessaris per esclarir la funcionalitat o forma d'implementació d'aquests fragments.

El component DronPrincipal possiblement és el que més càrrega de lògica de negoci tinga implementada. Aquest component s'encarrega de, per una banda rebre missatges dels altres drons, ja que és l'encarregat de vetllar per què aquests estiguen bé i de comunicar-se amb l'altre sistema en cas que fóra necessari. Per tant, per açò últim, també necessita poder enviar missatges. A més a més, és l'encarregat de produir els canvis necessaris al model reaccionant als missatges que li arriben des dels drons fills.

Per això, anem a utilitzar aquest component per definir diferents aspectes de la implementació dels nostres components, i dels altres comentarem sols els punts en els quals difereixen.

A.2.1 Declaració i anotació de la classe principal. Interfície pública

El primer aspecte important és el component en si. Per poder fer que una simple classe Java siga interpretada per Kevoree com un dels seus components, hem de definir-la correctament perquè aquest siga capaç de reconéixer el nostre projecte.

És per això que cadascun dels nostres projectes comptarà amb una classe principal, que li donarà nom al component. En aquest cas, la classe s'anomenarà **DronPrincipal**, i la definirem de la següent manera:

```
1 @ComponentType
2 public class DronPrincipal implements DronListener
```

Codi Font A.1: Declaració de la classe DronPrincipal

- @ComponentType: Aquesta anotació és pròpia de Kevoree. Serveix per indicar que aquesta classe és una classe que definirà un component Kevoree.
- implements DronListener: DronListener és una interfície per permetre la comunicació amb el component. En concret, servix per a què el fil que arranca el nostre component (per realitzar el comportament simulat) puga comunicar-se amb el nostre component per poder executar les accions necessàries en arribar qualsevol tipus d'esdeveniment. Aquesta interfície es declara de la següent manera:

```
1 public interface DronListener
```

Codi Font A.2: Declaració de la classe DronListener

I contindrà totes les declaracions dels mètodes que volem fer accessibles en el nostre component, en aquest cas:

```
1  void helloDron(DronMessage helloValue);
2
3  void consumeDron(Object o);
4
5  String getIdDron();
6
7  String getRolDron();
8
9  Boolean getEsPrincipal();
10
11 Boolean conexionActiva();
12
13 void obtenirLlistaDrons();
14
15 void executaScript(String componentScript);
```

Codi Font A.3: Classe DronListener

A.2.2 Annotacions rel·lacionades amb el cicle de vida de Kevoree

Altres anotacions que han de tindre tots els components de Kevoree són les que indiquem a continuació. Aquestes fan referència al cicle de vida dels models de Kevoree, que bàsicament és Start - Update - Stop:

- @Start: Amb aquesta anotació marquem el mètode que es llançarà quan el model s'aplique i el nostre component arranque.
- @Stop: Amb aquesta anotació marquem el mètode que es llançarà quan el model s'aplique i el nostre component es detinga (per exemple, en cas que l'eliminem del model).
- @Update: Amb aquesta anotació marquem el mètode que es llançarà quan el model s'actualitze i el nostre seguisca formant part del model.

En el nostre component, aquestes anotacions estan en els següents mètodes:

```
@Start
public void start() throws InterruptedException {

if (dronThread == null) {
HandlerDronEvents handler = new HandlerDronEvents();
Thread.setDefaultUncaughtExceptionHandler(handler);
dronThread = new DronThread();
dronThread.setDronListener(this);
dronThread.setIdDron(this.idDron);

if (this.esPrincipal == true) {
this.model = modelService.getCurrentModel();
this.localModel = this.cloner.clone(model.getModel());
this.funcionsDronPrincipal = new FuncionsDronPrincipal(
kevScriptService, modelService, context, localModel);
}

dronThread.start();
}

}

@Stop
public void stop() {
this.dronThread.setPararFil(true);
}

@Update
public void update() throws InterruptedException {
stop();
start();
}
```

Codi Font A.4: Mètodes start, stop i update, que donen resposta al cicle de vida de Kevoree.

El nostre mètode anotat amb @Start produirà l'arrancada del component quan el model s'haja aplicat amb èxit. Bàsicament ací el que farem serà arrancar el fil associat al component, que serà l'encarregat de realitzar la simulació del comportament del sistema i inicialitzar tots els elements necessaris per treballar amb el model. Aquests seran enviats com a referències en memòria a la classe FuncionsDronPrincipal, que després detallarem, però bàsicament serà una classe que s'encarregarà d'encapsular tot el comportament d'aplicació de nous models.

El mètode anotat amb @Stop simplement s'encarrega de matar el fil associat al component.

I per últim, el mètode anotat amb @Update, s'encarregarà de detenir i arrancar de nou el nostre component.

A.2.3 Paràmetres Kevoree

Seguidament, mostrarem alguns dels atributs que hem necessitat declarar per poder obtenir un correcte funcionament del component. Per una banda, hem utilitzat atributs propis del component, que es comporten com a inputs propis del component al model, i que si, també podem modificar en temps d'execució. Perquè Kevoree ho interprete com un atribut propi i no un mes de Java, s'ha d'anotar de la següent forma:

```
1 @Param(optional = false)
2 private String idDron;
```

Codi Font A.5: Variable idDron.

En aquest cas, l'anotació "optional"a fals vol dir que és un atribut que podem deixar en blanc.

A.2.4 Input i output. Ports d'entrada i eixida. Part essencial en la modificació del model en temps d'execució

Altre dels elements que necessitem és un port d'eixida per poder enviar missatges. Ho declarem de la següent forma:

```
1 @Output(optional = true)
2 private Port outputPort;
```

Codi Font A.6: Objecte outputPort, port d'eixida dels components.

En aquest cas, l'anotació "optional"a fals indica que el port d'eixida pot estar connectat o no a un canal de comunicació. En cas que no fóra opcional (és a dir, "optional"= false), en intentar aplicar un model amb un component d'aquest tipus sense enllaçar el port d'eixida a un canal provocaria una excepció i el model no s'aplicaria.

També necessitem per alguns dels nostres components un port d'entrada. Per això, utilitzem l'anotació també pròpia de Kevoree, @Input. Al nostre codi, apareix en el següent fragment de codi:

```
1 @Input(optional = true)
2 public void consumeDron(Object o) {
```

```
3   if (o instanceof String) {
4
5   Boolean averia = false;
6   String msg = (String) o;
7   StringToDronMessage stdm = new StringToDronMessage();
8   DronMessage message = stdm.toDronMessage(msg);
9
10  if (message.getEvent().equals(DroneEvents.AVERIA_CAMERA)) {
11  averia = true;
12
13  String componentScript = "remove node0." + message.getIdDron()
14  + " add node0.DronTrafi" + contadorIdDrons
15  + " : DronTrafic/1.1-SNAPSHOT set node0.DronTrafi"
16  + contadorIdDrons + ".idDron='DronTrafi"
17  + contadorIdDrons + "' start node0.DronTrafi"
18  + contadorIdDrons + " bind node0.DronTrafi"
19  + contadorIdDrons + ".outputPort SyncBroad1";
20
21  contadorIdDrons++;
22
23  this.executaScript(componentScript);
24
25  } else if (message.getEvent().equals(DroneEvents.BATERIA_BAIXA
        )) {
26
27  String componentScript = "remove node0." + message.getIdDron()
28  + " add node0.DronVigi" + contadorIdDrons
29  + " : DronVigilancia/1.1-SNAPSHOT set node0.DronVigi"
30  + contadorIdDrons + ".idDron='DronVigi"
31  + contadorIdDrons + "' start node0.DronVigi"
32  + contadorIdDrons + " bind node0.DronVigi"
33  + contadorIdDrons + ".outputPort SyncBroad1";
34
35  contadorIdDrons++;
36
37  this.executaScript(componentScript);
38
39  } else if (message.getEvent().equals(DroneEvents.
        DETECTAT_INCENDI)) {
40  this.helloDron(message);
41  } else {
42
43  System.out.println("MISSATGE REBUT PER: " + this.idDron);
44  System.out.println("[EMISSOR]: " + message.getIdDron());
45  if (message.getRol() != null)
46  System.out.println("[ROL EMISSOR]: "
47  + message.getRol().toString());
48  if (message.getEvent() != null)
```

```
49  System.out.println("[EVENT]: "
50  + message.getEvent().toString());
51  System.out.println("[MISSATGE]: " + message.getMissatge());
52
53  }
54
55  }
56  }
```

Codi Font A.7: Mètode consumeDron per respondre davant dels missatges rebuts.

Detallarem aquest codi per parts. Primerament tenim l'anotació @Input. Açò vol dir que qualsevol cosa que arribe pel port d'entrada d'aquest component entrarà per aquest mètode. En aquest cas, els missatges d'altres drons.

Per altra banda, la funcionalitat. Bàsicament el que fa aquest fragment de codi és transformar el String que ens arriba a un objecte de tipus DronMessage. Una vegada tenim l'objecte DronMessage, podem identificar-lo i actuar en conseqüència.

El que fem és diferenciar quin tipus de missatge està arribant. Per exemple, a la línia 10 identifiquem que el missatge que ens arriba és del tipus **AVARIA_CAMERA**. És a dir, el dron que ens ha enviat aquest missatge sofreix una avaria i deu ser substituït. Quan entrem a aquest fragment de codi, llancem un script en KevScript que el que fa és reemplaçar aquest component Dron per un altre d'equivalent i aplicar el canvi del model en temps d'execució, com podem veure a les línies 11-23. La resta de "else ifçontemplen altres tipus d'esdeveniments.

En cas de no identificar cap esdeveniment que requerisca un canvi, entrarem pel bloc "else", que l'únic que fa és mostrar per pantalla el missatge rebut.

A.2.5 Serveis Kevoree

Altres dels elements que necessitem són els següents:

```
1  @KevoreeInject
2  KevScriptService kevScriptService;
3
4  @KevoreeInject
5  ModelService modelService;
```

Codi Font A.8: Objectes kevScriptService i modelService.

Aquests elements són classes que ens proporciona Kevoree per accedir al que és el nucli d'aquest. Aquestes classes ens ofereixen una sèrie de serveis relacionats amb Kevoree i amb el treball amb models d'aquest. Utilitzem l'anotació @KevoreeIn-

ject perquè Kevoree inicialitze aquests serveis, ja que si no ens arribarien nuls, i no podríem treballar amb ells. És el mateix que realitza Spring amb l'anotació @Autowired.

- KevScriptService: Aquesta classe de Kevoree recull tots els serveis relacionats amb el treball amb KevScript. Ofereix serveis per executar fragments de codi en format KevScript, entre d'altres.
- ModelService: Aquesta classe de Kevoree recull tots els serveis relacionats amb el treball amb models. És la classe de Kevoree més important que anem a utilitzar, perquè, entre d'altres, ens ofereix serveis com el d'obtenir el model actual amb el mètode *getCurrentModel()* o d'aplicar el nostre model, amb l'ordre: *update(modelAAplicar, UpdateCallback())*

```
1
2  this.modelService.getCurrentModel().getModel()
3
4  this.modelService.update(localModel, new UpdateCallback()
       {
5
6                              public void run(Boolean arg0) {
7
8
9                              }
10                 });
```

Codi Font A.9: Codi per actualitzar el model.

A.2.6 Classes Kevoree per treballar amb models

Seguidament, mostrarem els atributs que hem declarat per poder treballar de forma adequada amb els models de Kevoree. El que hem d'entendre per poder assimilar bé el codi és la forma de treballar de Kevoree amb els models. Per resumir-ho, perquè ja ho hem comentat en l'apartat corresponent, podríem dir que Kevoree treballa amb un model que està corrent actualment (representat per UUIDModel). Per modificar-lo, hem de fer un clon d'aquest (amb l'ajuda de KMFFactory i ModelCloner), fer-li les modificacions que necessitem, i posteriorment aplicar-lo.

```
1
2  ContainerRoot localModel;
3
4  UUIDModel model;
5
6  private KMFFactory factory = new DefaultKevoreeFactory();
7
```

```
8 ModelCloner cloner = new ModelCloner(factory);
```

Codi Font A.10: Variables localModel, model, factory i cloner.

- ContainerRoot: Aquest objecte Kevoree serveix per guardar un model complet. Ens és molt útil per guardar ací la còpia del model que està actualment definint el sistema i poder treballar amb ell.
- UUIDModel: Aquest objecte Kevoree ens servirà per guardar una referència al model que s'està executant en Kevoree. Aquest model sempre serà de soles lectures.
- KMFFactory: Aquest objecte Kevoree és, com el seu propi nom indica, una espècie de factoria de models, ens ajuda a duplicar, crear models...
- ModelCloner: Aquest objecte Kevoree, treballant conjuntament amb l'objecte KMFFactory, ens ajuda a realitzar una còpia del model que vulguem. Molt necessari per clonar el model que està actualment corrent per poder aplicar-li els canvis que necessitem.

A.3 Codi font. Fil associat al Dron de Tràfic

Com hem comentat durant tot el treball, el nostre treball té una forta càrrega de simulació perquè no podem implementar el nostre sistema sobre elements físics. Per a dotar al sistema d'una execució continua que ens proporcione aquest aspecte de simulació, arranquem una sèrie de fils que s'encarreguen de simular aquest comportament.

El fil associat al component DronPrincipal no té pràcticament càrrega de lògica. L'únic que ens permet és obtenir un monitoratge constant de sistema. En canvi, el fil associat als altres drons sí que tenen més carrega de simulació del sistema. Ens hem decantat per mostrar la funcionalitat associada al fil que s'arranca amb el Dron de control de tràfic. Si s'analitza el fil associat al Dron de vigilància, podrem observar que les diferències són mínimes.

A.3.1 Variables necessàries

Per treballar amb aquest fil i tindre un comportament adequat amb el qual busquem, hem declarat les següents variables:

```
1 DronListener dronListener;
2
3 private String idDron;
```

```
4
5  private Boolean pararFil = false;
```

Codi Font A.11: Variables dronListener, idDron i pararFil.

- dronListener: Aquest objecte és la implementació de la interfície publica del dron associat al fil, en aquest cas, el dron de control de tràfic. Ens permetrà accedir des del fil als mètodes del component.
- idDron: Aquesta cadena serà l'identificador del dron associat al fil.
- pararFil: Aquesta variable Boolean és un flag que ens servirà per detenir el fil en cas que el component arribe al final del seu cicle de vida.

Totes aquestes variables són inicialitzades des del component una vegada es declara i s'arranca en fil durant la fase @Start del cicle de vida de Kevoree.

A.3.2 Arrencada del fil i procés de simulació

Per arrancar el fil que simularà el comportament del dron de control de tràfic, sobreescriurem el mètode run() de la següent forma:

```
1  public void run() {
2
3  try {
4  this.accionsTrafic();
5  } catch (InterruptedException e) {
6  e.printStackTrace();
7  } catch (Exception e) {
8  e.printStackTrace();
9  }
10
11 }
```

Codi Font A.12: Funció run per arrencar el fil associat al component.

De forma que simplement el que farà, quan el component arranque, és llançar una crida al mètode accionsTrafic() que és el que vertaderament conté la lògica del nostre fil. Aquest mètode és el següent:

```
1  public void accionsTrafic() throws Exception {
2
3  int contador = this.numeroRandom();
4
```

```
5   while (!this.pararFil) {
6
7   if (contador == 5) {
8   DronMessage missatge = new DronMessage();
9   missatge.setEvent(org.kevoree.dron.utils.DroneEvents.
        INTERCEPTAT_INFRACTOR_TRAFIC);
10  missatge.setIdDron(String.valueOf(this.getId()));
11  missatge.setRol(DronRoles.CAMERA_TRAFIC);
12  missatge.setMissatge(getInfraccioRandom());
13
14  Thread.sleep(12000);
15  dronListener.helloDron(missatge);
16
17  } else if (contador == 10) {
18  DronMessage missatge = new DronMessage();
19  missatge.setEvent(org.kevoree.dron.utils.DroneEvents.
        AVERIA_CAMERA);
20  missatge.setIdDron(this.idDron);
21  missatge.setRol(DronRoles.CAMERA_TRAFIC);
22  missatge.setMissatge("Camera averiada. Substituir dron.");
23
24  this.dronListener.helloDron(missatge);
25  this.pararFil = true;
26  } else {
27  DronMessage missatge = new DronMessage();
28  missatge.setEvent(org.kevoree.dron.utils.DroneEvents.
        INTERCEPTAT_INFRACTOR_TRAFIC);
29  missatge.setIdDron(this.idDron);
30  missatge.setRol(DronRoles.CAMERA_TRAFIC);
31  missatge.setMissatge("[MATRICULA]= " + getMatriculaAleatoria()
32  + "[VELOCITAT]= " + getVelocitatRandom());
33
34  Thread.sleep(8000);
35  dronListener.helloDron(missatge);
36  }
37
38  }
39  }
```

Codi Font A.13: Classe accionsTrafic, lògica del fil associat al Dron de control de tràfic.

Aquest és un mètode per realitzar la simulació, com ja hem comentat, bastant simple, que bàsicament, s'encarrega de fer el següent:

- Obtenim un nombre aleatori i el guardem en la variable çontador". Aquest nombre ens servirà després per decidir sobre el comportament del component.

- Si i sols si la variable "pararFil"no és certa, entrarem a la part central del mètode per simular el comportament. Aquesta variable, com hem nomenat anteriorment, s'estableix a fals en cas que hi haja un canvi al model que provoque una parada del component.
- Una vegada al bucle, es poden donar 3 circumstàncies.
  - Que la variable çontador"siga igual a cinc. En aquest cas, el nostre component enviarà un missatge a través del mètode "helloDron"(port d'eixida) amb les dades d'un infractor de tràfic que ha comés una infracció de tipus aleatori.
  - Que la variable çontador"siga igual a deu. En aquest cas, també s'enviarà un missatge però indicant que la càmera del dron ha sofert una avaria i s'ha de canviar, la qual cosa provocarà que el dron principal modifique el model i substituïsca aquest dron per un altre.
  - Que la variable çontador"siga igual a qualsevol altre nombre. En aquest cas es llançarà un missatge, el més comú, d'una infracció de tràfic associada a un excés de velocitat.
- En qualsevol dels tres casos, el fil dorm durant un nombre definit de mili segons, i posteriorment i si la variable "pararFil"segueix sent vertadera, torna a iterar al bucle.

Aquest és el gros de la simulació dels nostres components. Ara veurem com treballen els semàfors i en concret el coordinador principal d'aquests, que serà l'encarregat de produir els canvis al model.

A.4 Coordinador Semàfors

Mentre que el nostre sistema de drons basa la seua intercomunicació en enllaços via canals de Kevoree i l'enviament i recepció de missatges, el nostre subsistema de semàfors basa aquesta comunicació en canvis de model. És a dir, en el sistema de drons són els drons, diguem, que formen la part més inferior del sistema els que es comuniquen amb el dron principal via enviament de missatges. Però, en el sistema de semàfors, és el coordinador de semàfors el que es comunica amb els semàfors que estan a un nivell més baix, però no ho fa mitjançant missatges ni canals de comunicació, sinó que el que fa és produir un canvi en el model que afecta directament a aquests semàfors. És a dir, per posar un exemple clarificador, en compte de què el coordinador envie un missatge al semàfor A dient-li "posa't en roig!"és directament aquest coordinador el que canvia la propietat adequada del semàfor en el model i, seguidament, aplica aquests canvis al model que s'està executant.

```
1  public void invertirSemafors() {
2
3          List<String> components = this.obtindreTotsSemafors();
4
5          StringBuilder sb = new StringBuilder();
6
7          for (String semafor : components) {
8                  if (semafor.toLowerCase().equals("semafor1"))
                       {
9                          sb.append("set node0.semafor1.estat='"
                               + this.estat1).append("' ");
10                 } else {
11                         sb.append("set node0.semafor2.estat='"
                               + this.estat2).append("' ");
12                 }
13         }
14
15         String aux = estat2;
16         estat2 = estat1;
17         estat1 = aux;
18
19         this.executaScript(sb.toString());
20
21         contador++;
22
23 }
```

Codi Font A.14: Funció invertirSemafors, per alterar el estat dels components de tipus semàfor.

Aquest fragment de codi fa precisament açò que hem comentat, provocar un canvi a la propietat 'estat' associada als semàfors del nostre sistema i aplicar el canvi del model al model que s'està executant. En aquest cas, com és una simulació, assumim que el nostre sistema té dos semàfors, però es podria ampliar a N perfectament.

No entrarem en detall en les classes i objectes que declarem perquè bàsicament són les mateixes que utilitzem als components de tipus dron.

A.5 Semàfor

El component semàfor no compta amb una lògica pròpia, sinó que depén dels canvis al model produïts pel component CoordinadorSemàfors. És per això que s'inclou el codi font a aquesta documentació.

A.6 Paquet Utils. Reutilització de codi.

Per a facilitar el treball amb els nostres components hem definit un altre projecte Maven a banda. Aquest el que fa és definir una sèrie de classes amb elements comuns al sistema (constants, mètodes...) i és importat i utilitzat per quasi tots els components que formen el sistema tant de drons com de semàfors.

L'estructura que presenta el nostre paquet dronUtils és la següent:

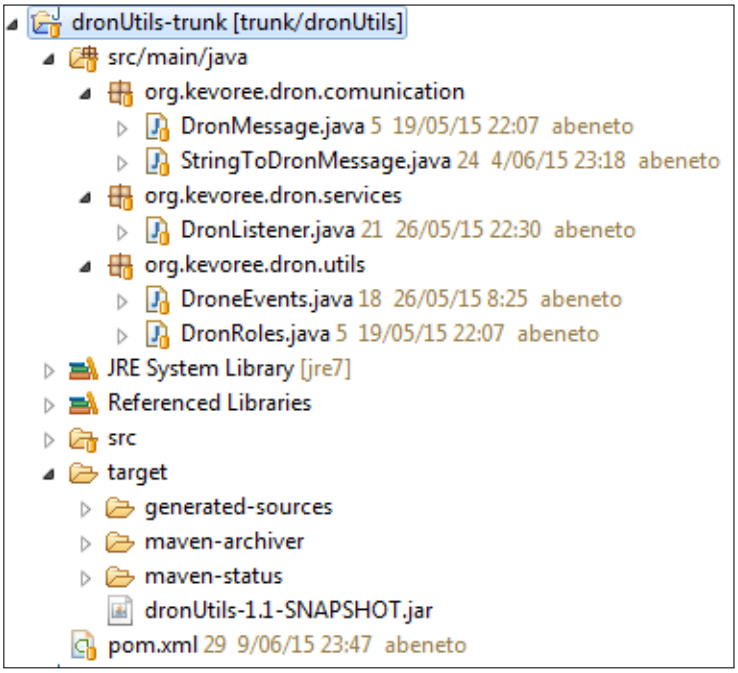

Figura A.2: Estructura del paquet dronUtils.

Com podem veure dins de dronUtils hi ha tres paquets Java amb les seues respectives classes. Aquests són:

- org.kevoree.comunication: Aquest paquet recull funcionalitat relacionada amb la comunicació de components. Les classes que conté són:

  - DronMessage: Aquesta classe representa els objectes de tipus missatge que s'enviaran pels canals de comunicació del nostre sistema. Bàsicament conté quatre elements per definir el nostre missatge:

    * DronEvents event: Esdeveniment que intenta comunicar el component
    * DronRoles rol: Rol del component que envia el missatge.
    * String idDron: Identificador del component que envia el missatge.
    * String missatge: Missatge que envia el component.

```
1  public class DronMessage {
2
3          private DroneEvents event;
4          private DronRoles rol;
5          private String idDron;
6          private String missatge;
7
8          public DronMessage() {
9                  super();
10         }
11
12 public DronMessage(DroneEvents event, DronRoles rol
       , String idDron, String missatge) {
13                 super();
14                 this.event = event;
15                 this.rol = rol;
16                 this.idDron = idDron;
17                 this.missatge = missatge;
18         }
```

Codi Font A.15: Classe DronMessage

  - StringToDronMessage: Aquesta classe conté els mecanismes per poder transformar una cadena de caràcters en un objecte de tipus DronMessage. Necessitem aquesta classe pel fet que Kevoree sols permet enviar objectes de tipus String pels seus canals de comunicació, i, quan arriben al seu destinatari, necessitem reconstruir-los per poder treballar més còmodament amb ells. L'únic mètode que conte aquesta classe és el que ens dóna aquesta funcionalitat i és aquest:

```
1  public DronMessage toDronMessage(String missatge) {
2  DronMessage droneMessage = new DronMessage();
```

```
3
4   String array[] = missatge.split(":");
5
6   if(array[0].contains(DroneEvents.AVERIA_CAMERA.
        toString())){
7           droneMessage.setEvent(DroneEvents.
                AVERIA_CAMERA);
8   } else if (array[0].contains(DroneEvents.AVERIA_HELIX.
        toString())) {
9           droneMessage.setEvent(DroneEvents.AVERIA_HELIX
                );
10  } else if (array[0].contains(DroneEvents.BATERIA_BAIXA
        .toString())) {
11          droneMessage.setEvent(DroneEvents.
                BATERIA_BAIXA);
12  } else if (array[0].contains(DroneEvents.
        INTERCEPTAT_INFRACTOR_TRAFIC.toString())) {
13          droneMessage.setEvent(DroneEvents.
                INTERCEPTAT_INFRACTOR_TRAFIC);
14  } else if (array[0].contains(DroneEvents.
        SOBREPASAT_PUNT_NO_RETORN.toString())) {
15          droneMessage.setEvent(DroneEvents.
                SOBREPASAT_PUNT_NO_RETORN);
16  } else if (array[0].contains(DroneEvents.
        DETECTAT_INCENDI.toString())) {
17          droneMessage.setEvent(DroneEvents.
                DETECTAT_INCENDI);
18  }
19
20  if(array[1].contains(DronRoles.CAMERA_INCENDIS.
        toString())) {
21          droneMessage.setRol(DronRoles.CAMERA_INCENDIS)
                ;
22  } else if(array[1].contains(DronRoles.CAMERA_TRAFIC.
        toString())) {
23          droneMessage.setRol(DronRoles.CAMERA_TRAFIC);
24  } else if(array[1].contains(DronRoles.VIGILANCIA.
        toString())) {
25          droneMessage.setRol(DronRoles.VIGILANCIA);
26  }
27
28  droneMessage.setIdDron(array[2]);
29  droneMessage.setMissatge(array[3]);
30
31  return droneMessage;
32  }
```

Codi Font A.16: Funció toDronMessage, per transformr una cadena en un objecte DronMessage

- org.kevoree.dron.services: Aquest paquet conté la interfície que tots els drons utilitzaran per fer publics els seus mètodes al fil associat a cadascun d'ells.
- org.kevoree.dron.utils: Aquest paquet conté dues classes de tipus Enum que serveixen per restringir i enumerar els tipus de Rols i Esdeveniments que pot fer servir un component. En concret
  - DronEvents: Recull els esdeveniments que pot utilitzar un component.
  - DronRoles: Recull els rols que pot adoptar un component.

Quan compilem aquest paquet en Maven, l'artefacte que obtindrem serà el fitxer *dronUtils-versio.jar*, que serà la llibreria importada a la resta de components del sistema. Per compilar-los en Maven, simplement haurem d'executar l'ordre:

```
1 mvn install
```

Codi Font A.17: Ordre per compilar un projecte en Maven.

Aquesta ordre la podem executar directament des de Eclipse, definint-la com una "external tool", o des de línia de comandaments.

A.7 Passes per arrancar Kevoree

Per arrancar Kevoree comptem amb dos elements essencials: l'entorn d'execució i l'editor Kevoree. L'entorn d'execució serà la base del nostre sistema, on correrà aquest, representat pels models que definim. Aquests models els definirem a l'editor.

L'entorn d'execució el podem descarregar de http://kevoree.org/download.html. L'opció que hem d'escollir és "Java SE Runtime". L'editor també el descarregarem d'aquest mateix URL, des de l'opció "Editor".

Una vegada tenim els dos jar descarregats, simplement ens queda arrancar-los. Per això executarem les següents ordres:

java -jar org.kevoree.platform.standalone-5.2.5.jar

java -jar org.kevoree.tools.ui.editor-5.2.5.jar.

Hem de tenir en compte que l'última versió estable tant de l'entorn d'execució com de l'editor és la 5.2.5, encara que molts dels components que ens oferix Kevoree es troben en versions més avançades.

Una vegada llançats els dos comandaments i si no s'ha produït cap problema, tindrem el runtime arrancat i l'editor en marxa. Passarem ara al següent punt, que ens indicarà com arribar al modelatge del sistema que hem definit per què el nostre prototip funcione.

A.8 Passes per modelar el sistema

Una vegada arrancat l'entorn d'execució i l'editor, hem de procedir a importar les llibreries que inclouen els components necessaris per modelar el nostre sistema. Per una banda, hem d'importar les llibreries que ens permetran incloure canals de comunicació i, per altra banda, les llibreries associades als components que hem generat.

Però el primer que hem de fer i més important és carregar el model del runtime que ja està en execució. Aquest model compta simplement amb un component WSGroup enllaçat a un node, el node0. Aquest és sempre el runtime bàsic de Kevoree sobre el que construirem els nostres models. Per fer-ho, accedirem a la secció del menú "File > Open from Node". Una vegada que polsem ací, se'ns obrirà una xicoteta finestra en la qual ens preguntarà on està corrent el nostre runtime. Com l'hem alçat en el nostre ordinador local i el port sempre és el 9000, ho deixarem tal com apareix en la imatge.

Per importar llibreries, hem de accedir a l'opció de menú "Model > Load Kevoree Libraries > Java". Ací dins se'ns obrirà un extens llistat. Cadascuna d'aquesta llibreria inclou, normalment, més d'un component. Afegirem les dues llibreries que apareixen marcades a la imatge i que són les que ens interessen:

org.kevoree.library.java.javaNode org.kevoree.library.java.channels,

Aquestes llibreries ens proporcionen els components de tipus Node (els contenidors dels components) i de tipus Channel (vies de comunicació entre components).

Una vegada fet açò, veurem que els components inclosos en aquestes llibreries apareixen a la part esquerra de la pantalla, que és on apareixeran tots els components que importem i, per tant, amb els que podem treballar per definir el nostre model.

I amb açò ja ho tenim pràcticament fet, el que ens falta és feina simplement d'arrossegar i soltar. Això si, tenint prèviament unes consideracions de com hem de modelar el nostre sistema perquè funcione bé. Per una banda, no importa si els nostres elements estan tots dins del mateix node o de diversos, però si estan repartits en diversos nodes com hem mostrat en el plantejament del punt 4.3.2, han

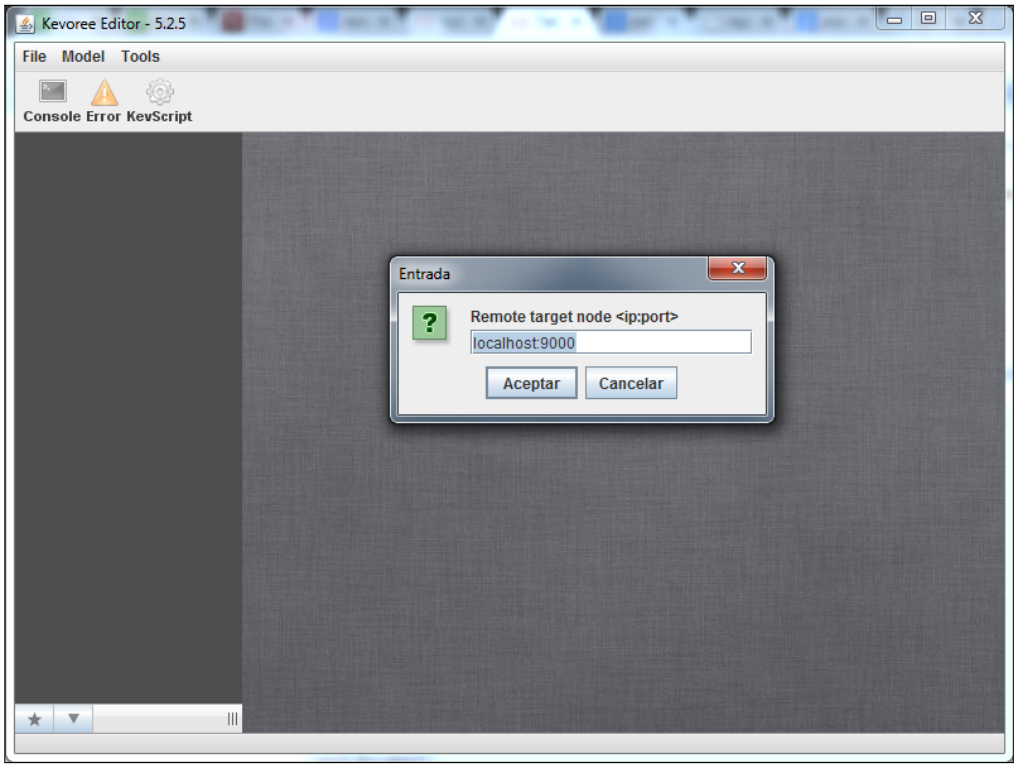

Figura A.3: Carregar el model vinculat al runtime Kevoree en el nostre equip local sobre el port 9000

d'estar continguts dins d'un node principal, el node0. Açò és a causa dels problemes de comunicació que presenta Kevoree i que ja hem detallat en els apartats 4.4 i 4.6. Per facilitar la seua comprensió, aquest prototip el dissenyarem sobre un únic node.

Bàsicament, el nostre model ha de complir els següents requisits.

- Ha de comptar amb un element de tipus CoordinadorSemafor i DronPrincipal. A més, han d'estar intercomunicats (eixida del DronPrincipal a entrada del CoordinadorSemafor) perquè el DronPrincipal puga transmetre informació al CoordinadorSemafor.
- Ha de comptar amb dos elements de tipus semàfor, que tindran com a identificador semafor1 i semafor2, i els seus estats d'origen seran roig i verd

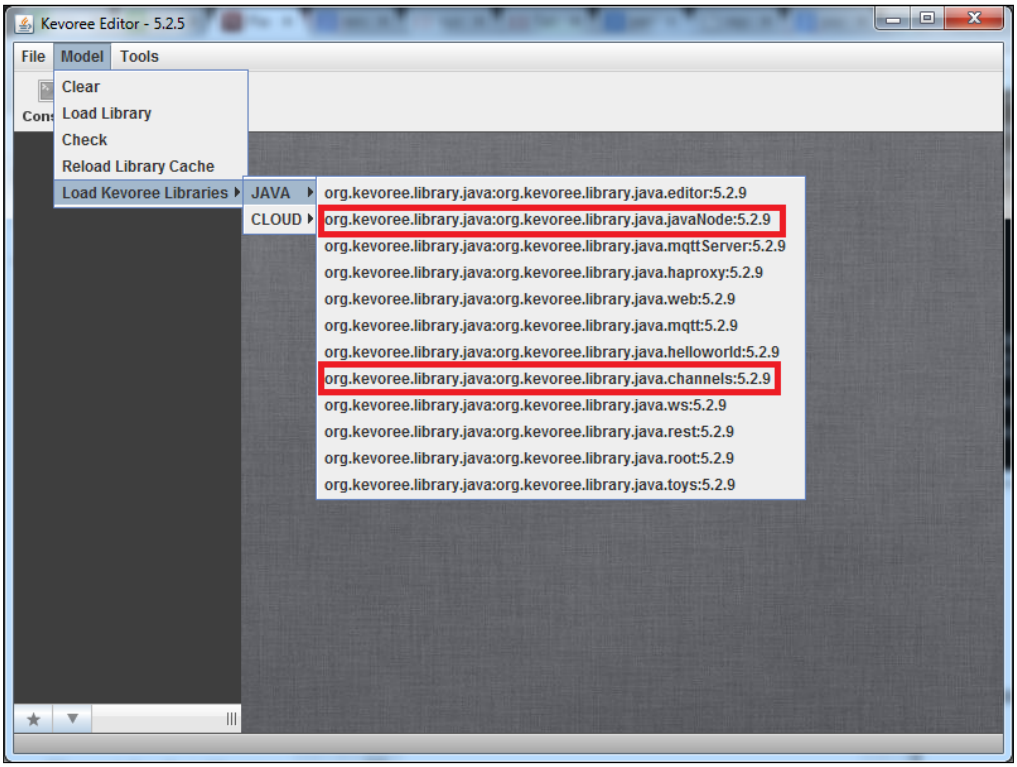

Figura A.4: Llibreries necessàries per treballar amb el nostre prototip

respectivament. Açò ho podem configurar fent click damunt de l'objecte semàfor una vegada forma part del model. Quan fem click se'ns obri una finestra per afegir aquests atributs.

- No hi ha restricció pel que fa al nombre d'elements de tipus DronVigilancia i DronTrafic.
- A diferència dels elements de tipus semàfor, els quals no requereixen canals de comunicació, els elements de tipus DronTrafic i DronVigilancia sí que necessiten aquests canals per comunicar-se amb el DronPrincipal. De forma que afegirem un Channel del tipus SyncBroadCast i enllaçarem l'eixida d'aquests amb l'entrada del DronPrincipal

Ací podem veure el sistema completament modelat:

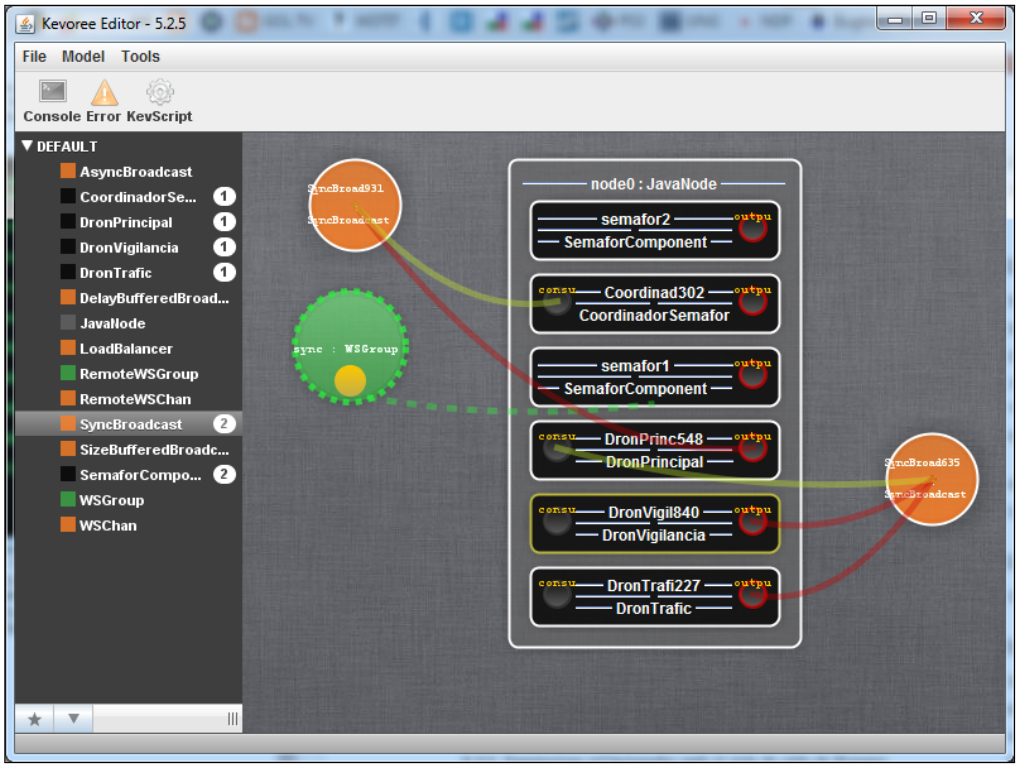

Figura A.5: Modelat complet del nostre sistema.

Per fer que el nostre model es transforme en un sistema real i en execució, hem de seguir els següents passos:

- Polsar damunt del component node0, que contindrà tots els elements del nostre sistema.
- Una vegada polsat, se'ns obrirà una finestra en la qual vorem un botó que posa "push"
- Polsar aquest botó.
- Mirarem la consola on hem arrancat el runtime Kevoree, que serà l'eixida dels esdeveniments que es produïsquen en el nostre sistema. Si una vegada polsat push veiem una linea que diu "End deploy result=true"tot ha anat correctament.

Una vegada que el nostre model està ja funcionant, podrem veure l'eixida que produeix aquest per línia de comandaments. Hem de deixar clara una cosa: encara que es produeixen canvis al model, si des de l'editor no recarreguem el model, no veurem aquests canvis de forma gràfica, encara que sí que podrem veure'ls reflectits a causa de l'eixida per línia de comandaments. Açò és una limitació pròpia de Kevoree.

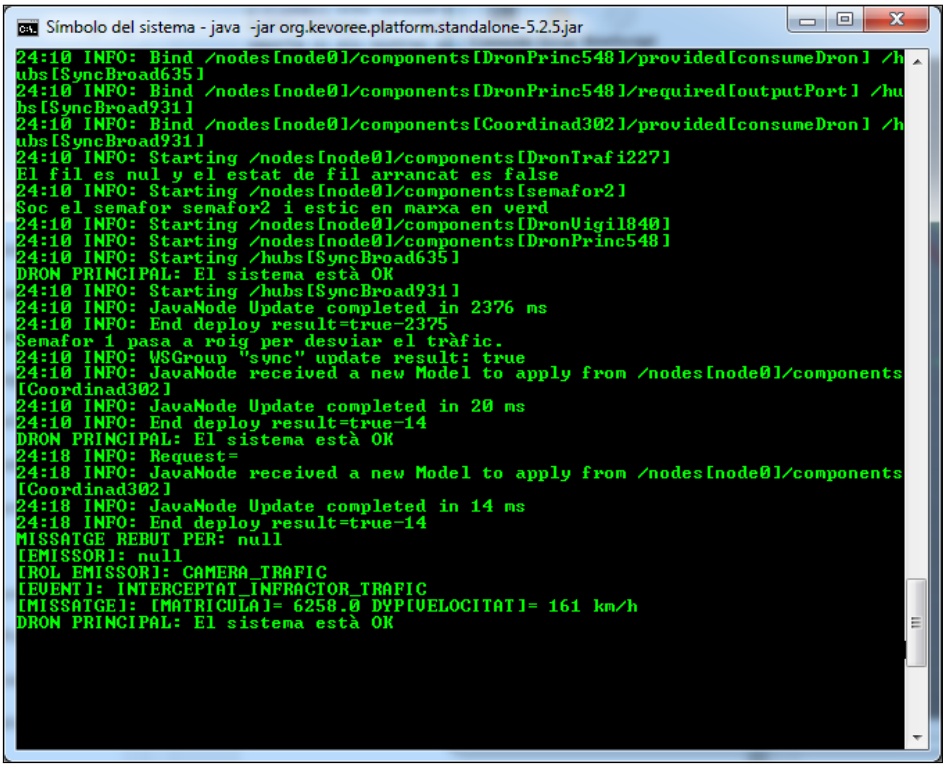

Figura A.6: Eixida per consola del nostre sistema en funcionament.

Bibliografia

Cabot, Jordi (2011). "MDD - Desarrollo de software dirigido por modelos que funciona (¡de verdad!)" A: *http://goo.gl/9UDyI5*, pàg. 6 (v. la pàg. 12).

Cetina Englada, Carlos (1998). *Achieving Autonomic Computing through the Use of Variability Models at Run-time*. Universitat Politècnica de València (v. la pàg. 11).

Fouquet,François et al. (2015a). "Learn Kevoree". A: *http://kevoree.github.io/kevoree-tutorials/*, (v. la pàg. 19).

— (2015b). "The Kevoree Book". A: *http://kevoree.github.io/kevoree-book/*, (v. la pàg. 19).

Gómez Lacruz, Maria (2011). *Designing Self-Adaptative Systems through Models at Run-time*. Universitat Politècnica de València (v. les pàg. 11, 12).

Kuecker, Glen David (2013). "Building the Bridge to the Future: New Songdo City from a Critical Urbanism Perspective". A: *DePauw University*, pàg. 1 - 3 (v. la pàg. 17).

Martínez Acuña, Manuel Ignacio (2013). "Inteligencia ambiental". A: *Revista de divulgación científica y tecnológica de la universidad veracruzana*, pàg. 6 (v. la pàg. 16).

Morin, Brice et al. (2009). "Models@Run.time to support dynamic adaptation". A: *www.irisa.fr/triskell/publis/2009/Morin09f.pdf*, pàg. 1 - 2 (v. la pàg. 14).

Pons, Claudia, Roxana Giandini i Gabriela Pérez (2010). *Desarrollo de Software Dirigido por Modelos*. Mc Graw Hill (v. la pàg. 14).